Prediction Trees

CART, Bagging, and Random Forest

DS 6030 | Fall 2024

trees.pdf

Contents

Some of the figures in this presentation are taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013) with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

1 Classification and Regression Tree Intro

Tree-based methods:

1. Partition the feature space into a set of (hyper) rectangles.
2. Fit a simple model (e.g., constant) in each region.

They are conceptually simple yet powerful.

- Main Characteristics:
  - flexibility, intuitive, non-model based
  - natural graphical display, easy to interpret
  - building blocks of Random Forest and (Tree-based) Boosting
  - naturally includes feature interactions
  - reduces need for monotonic feature transformations
- Main Implementations:
  - CART (Classification and Regression Trees) by Breiman, Friedman, Olshen, Stone (1984)
  - C4.5 Quinlan (1993)
  - Conditional Inference Trees (`party` R package)

1.1 Prediction Trees

Figure 1: House Pricing in Sacramento, CA

1.2 Building Prediction Trees

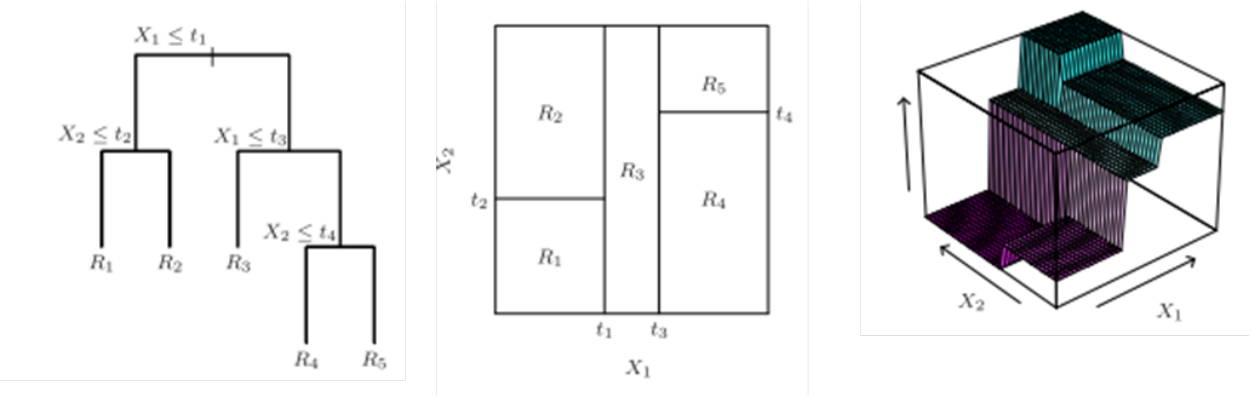
As usual, we want find the trees that make predictions which minimizes some loss function.

- **Classification trees** have class probabilities at the leaves (e.g., the probability of heavy rain is 0.9).
  - E.g., Loss = Negative Binomial likelihood.
- **Regression trees** have a mean response at the leaves. (e.g., the expected amount of rain is 2in).
  - E.g., Loss = Mean squared error.

1.3 Recursive Binary Partition (CART)

Because the number of possible trees is too large to exhaustively search, we usually restrict attention to *recursive binary partition trees* (CART).

- These are also easy to interpret



Think of the *reverse* of agglomerative hierarchical clustering

- In hierarchical clustering, we started with all observations in clusters of size 1 and then sequentially grouped them together, according to some measure of *homogeneity/similiarity/distance/dissimilarity/loss*, until there was one big cluster.
  - The optimal clustering is usually somewhere between the two extremes
- In CART, all observations start in one big group and are split into two subgroups. Each subgroup is then split into two additional subgroups. This is repeated until some stopping criteria is met (e.g., not enough observations in to split further). The terminal subgroup (leaf nodes) are used to make predictions.
  - The splitting is also based on some measure of homogeneity/similarity/loss.
  - Since we are in a supervised setting, the splitting criterion should be based on how well the new groups estimate the outcome variable.
  - There is another important difference: in CART *only a single feature* is used to determine the split into subgroups.

1.3.1 Model and Model Parameters

Trees model the outcome as a *constant* in each region

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \hat{c}_m \, \mathbb{1}(\mathbf{x} \in \hat{R}_m)$$

- The *model parameters* of a tree, T, with M leaf nodes, are:
  - The regions (*leaf nodes*) $R_1, \ldots, R_M$
  - The coefficients/scores for the regions $c_1, \ldots, c_M$
- Given the regions, leaf-node coefficients are based on the choice of loss function
  - Under Squared Error (regression):

$$\begin{aligned}\hat{c}_m &= \text{Ave}(\{y_i : \mathbf{x}_i \in R_m\}) \\ &= \frac{1}{N_m} \sum_{i:\mathbf{x}_i \in R_m} y_i\end{aligned}$$

  - Under log-loss (soft classification), the coefficients are probability *vectors* (one element for each class; sums to one).

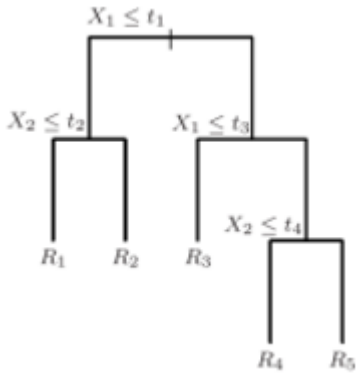
$$\begin{aligned}\hat{c}_{mk} &= \text{Proportion of class } k \text{ in region } R_m \\ &= \frac{1}{N_m} \sum_{i:\mathbf{x}_i \in R_m} \mathbb{1}(y_i = k)\end{aligned}$$

  - Under 0-1 loss (hard classification), the coefficients are one-hot vectors.

$$\begin{aligned}\hat{c}_{mk} &= \text{One hot for majority class} \\ &= \mathbb{1}(k \text{ is majority class in region } R_m)\end{aligned}$$

  - Other options possible; choose the coefficients to optimize your particular objective function. Note: check the loss (implicitly) used in growing the tree!

1.3.2 Basis Expansion Interpretation



$$f(x) = \sum_{m=1}^{M} \theta_m b_m(x)$$

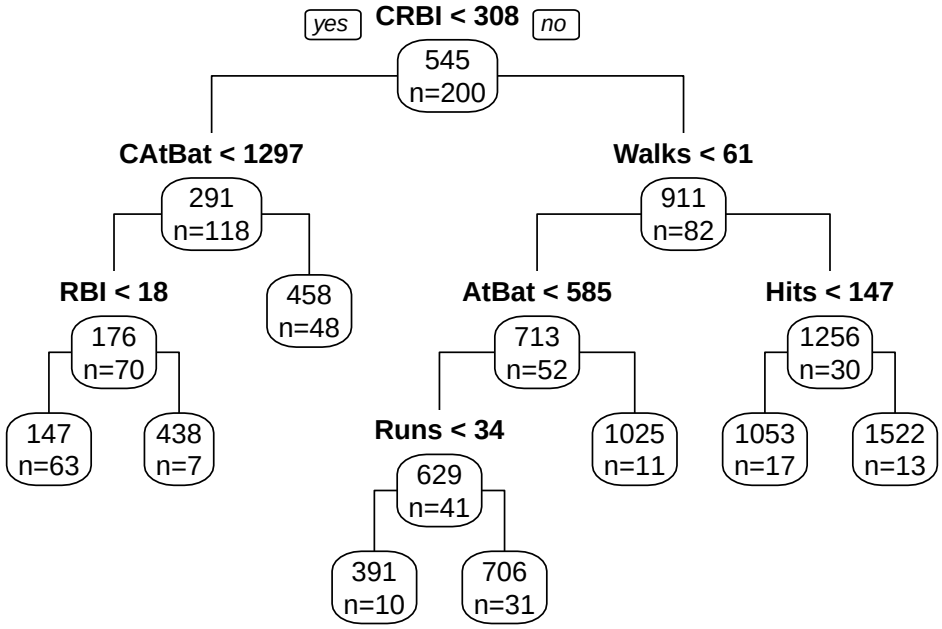
$$\begin{aligned}
R_1 &: b_1(x_1, x_2) = \mathbb{1}(x_1 \leq t_1)\, \mathbb{1}(x_2 \leq t_2) \\
R_2 &: b_2(x_1, x_2) = \mathbb{1}(x_1 \leq t_1)\, \mathbb{1}(x_2 > t_2) \\
R_3 &: b_3(x_1, x_2) = \mathbb{1}(x_1 > t_1)\, \mathbb{1}(x_1 \leq t_3) \\
R_4 &: b_4(x_1, x_2) = \mathbb{1}(x_1 > t_1)\, \mathbb{1}(x_1 > t_3)\, \mathbb{1}(x_2 \leq t_4) \\
R_5 &: b_5(x_1, x_2) = \mathbb{1}(x_1 > t_1)\, \mathbb{1}(x_1 > t_3)\, \mathbb{1}(x_2 > t_4)
\end{aligned}$$

1.3.3 Example: Baseball Salaries

The `ISLR` R package (corresponding to the ISLR textbook), contains data (`Hitters`) on Major League Baseball players for the 1986-1987 season.

```
data(Hitters, package='ISLR')
```

Here is a CART tree for predicting the salary (in thousands dollars):



Your Turn #1 : Tree Interpretation

1. How many leaves are on the tree?
2. What do the numbers in the boxes mean?
3. How could you evaluate the prediction in a leaf node?

1.4 Growing a Tree

CART uses a greedy algorithm to grow a tree.

- Split the feature space into two pieces and predict the outcome in each region
  - Find the predictor j (out of $1, 2, \ldots, p$) and split point t (from unique ordered values of X_j or categories) to minimize the *loss function*

  - Produces two regions:

$$R_1(j,t) = \{x : x_j \leq t\} \text{ and } R_2(j,t) = \{x : x_j > t\} \qquad \textit{Numeric/Ordered Feature}$$

or

$$R_1(j,t) = \{x : x_j \in A_j\} \text{ and } R_2(j,t) = \{x : x_j \notin A_j\} \qquad \textit{Nominal/Categorical Feature}$$

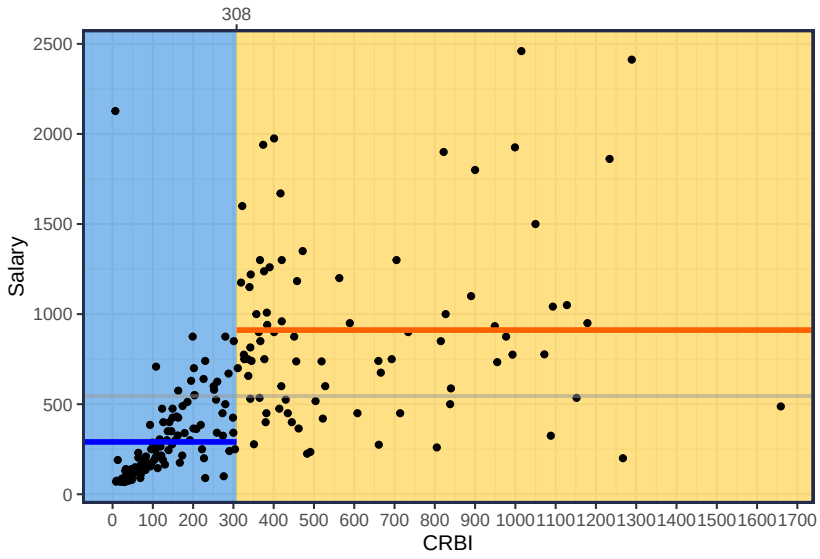
- Repeat this step for each **child** region
- Continue until stopping criteria met, e.g.
  - Minimum number of observations in region
  - Loss function has minimal improvement
  - Maximum depth (number of interactions)
- The final regions are called **leaf** nodes

1.5 Splitting Details

1.5.1 Regression Trees and Numeric Features

Notice in the fitted tree for the baseball data that the first split was based on a player's *Career RBIs* (`CRBI`). Specifically, if a player has less than `308` Career RBIs they go down the left side, otherwise they go down the right side.

Let's examine this first split:



- This is basically a univariate *change point model*
  - The split point (`CRBI` < 308) is the best change point (change in mean) using a Gaussian model
  - An alternative perspective is to see that the reduction in MSE/SSE is maximized by splitting at (`CRBI` < 308) and fitting the data on each side of the split with a constant.

Splitting Details: Squared Error Loss

Notation

- $y \in \mathbb{R}$
- $\mathbf{x} = [x_1, \ldots, x_p]^\mathsf{T}$
- n observations (in current node/region)

Consider a split on feature j:

- Before split, the quality of the model, based on SSE is:

$$Q_0 = \sum_{i=1}^{n}(y_i - \bar{y})^2 \qquad \text{where } \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

- Consider a split at s (on feature j)

Left Region

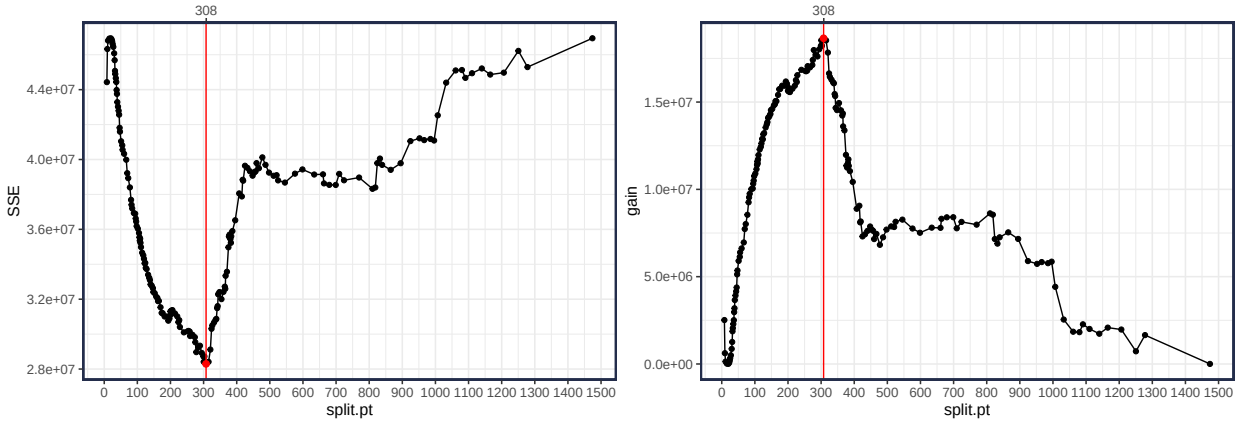
$$R_1(s) = \{x : x_j < s\}$$
$$\bar{y}_1(s) = \frac{1}{n_1} \sum_{\{i:x_i \in R_1(s)\}} y_i$$
$$Q_1(s) = \sum_{\{i:x_i \in R_1(s)\}} (y_i - \bar{y}_1(s))^2$$

Right Region

$$R_2(s) = \{x : x_j \geq s\}$$
$$\bar{y}_2(s) = \frac{1}{n_2} \sum_{\{i:x_i \in R_2(s)\}} y_i$$
$$Q_2(s) = \sum_{\{i:x_i \in R_2(s)\}} (y_i - \bar{y}_2(s))^2$$

- Updated SSE: $Q(s) = Q_1(s) + Q_2(s)$
- $\text{Gain}(s) = Q_0 - Q(s)$

We can examine the SSE (or Gain) for all possible split points:

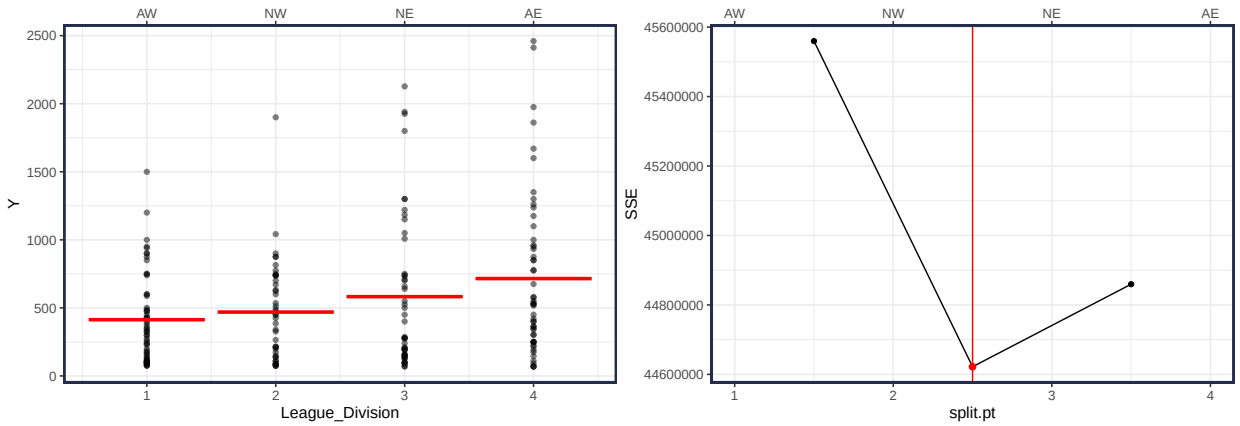


1.5.2 Regression Trees and Categorical (Nominal) Features

A categorical feature (with k levels) can be split into two groups $2^{k-1} - 1$ different ways.

- $k = 3$: 3 possible partitions
- $k = 4$: 7 possible partitions
- $k = 10$: 511 possible partitions

The CART approach sorts the categories by the mean response (recodes to numeric) and then splits like its a numeric feature.



Note

- Note: features with many levels will be split too often. Consider the quote from ESL (pg. 310)

  The partitioning algorithm tends to favor categorical predictors with many levels k; the number of partitions grows exponentially in k, and the more choices we have, the more likely we can find a good one for the data at hand. This can lead to severe overfitting if k is large, and such variables should be avoided.

- An alternative is to use one-hot-encoding to split a categorical feature into k new features.
  - As done by XGBoost
- There are other ways to *encode* categorical data so they can be treated like numeric (i.e., ordered data)
  - See e.g., CatBoost

1.5.3 Classification/Probability Trees

A **classification tree** (or probability tree) is a decision tree used for classifying categorical outcomes: $y \in \mathcal{G} = (1, 2, \ldots, K)$. The tree recursively partitions the feature space into regions and assigns each region a class label or probability vector representing the likely outcome of any point falling into that region.

- In region R_m, the probability of class k can be estimated as:

$$\begin{aligned}\bar{p}_m(k) &= \widehat{\Pr}(y = k \mid \mathbf{x} \in R_m) \\ &= \frac{1}{n_m} \sum_{\{i:\mathbf{x}\in R_m\}} \mathbb{1}(y_i = k) \\ &= \frac{n_{m,k}}{n_m}\end{aligned}$$

- Each region R_m is assigned a K-dimensional vector of estimated class probabilities, denoted by:

$$\bar{p}_m = [\bar{p}_m(1), \bar{p}_m(2), \ldots, \bar{p}_m(K)]$$

  where $\bar{p}_m(k)$ represents the probability estimate for class k in region R_m. Naturally, these probabilities must sum to 1:

$$\sum_{k=1}^{K} \bar{p}_m(k) = 1$$

There are three common measures of *node impurity* in this setting:

1. **Misclassification Error**: This measures the proportion of observations that do not belong to the majority class in region R_m. It is minimized when all observations in a node belong to the same class:

$$Q_m = 1 - \max_k \bar{p}_m(k)$$

2. **Gini Index**: The Gini Index measures the likelihood of misclassifying a randomly chosen observation from the node. It is 0 when all observations in a node belong to the same class and reaches its maximum when class probabilities are uniform.

$$\begin{aligned} Q_m &= \sum_{k=1}^{K} \bar{p}_m(k)(1 - \bar{p}_m(k)) \\ &= 1 - \sum_{k=1}^{K} \bar{p}_m^2(k) \end{aligned}$$

3. **Cross-Entropy/Deviance**: This measures the amount of uncertainty in the node. It is minimized when one class probability is 1 (i.e., full certainty about class membership) and is maximized when class probabilities are evenly distributed. It is proportional to the negative of the multinomial log-likelihood (and hence deviance).

$$\begin{aligned} Q_m &= -\sum_{k=1}^{K} \bar{p}_m(k) \log \bar{p}_m(k) \\ &= \sum_{k=1}^{K} \bar{p}_m(k) \log \frac{1}{\bar{p}_m(k)} \end{aligned}$$

1.5.3.1 Example:

Consider a node with three classes where the class probabilities are:

$$\bar{p}_m(1) = 0.5, \quad \bar{p}_m(2) = 0.3, \quad \bar{p}_m(3) = 0.2$$

Misclassification Error

$$\begin{aligned} Q_m &= 1 - \max_k \bar{p}_m(k) \\ &= 1 - \max(0.5, 0.3, 0.2) \\ &= 1 - 0.5 = 0.5 \end{aligned}$$

Gini Index

$$\begin{aligned} Q_m &= 1 - \left(\bar{p}_m(1)^2 + \bar{p}_m(2)^2 + \bar{p}_m(3)^2\right) \\ &= 1 - \left(0.5^2 + 0.3^2 + 0.2^2\right) \\ &= 1 - 0.38 = 0.62 \end{aligned}$$

Cross-Entropy

$$\begin{aligned} Q_m &= -\sum_{k=1}^{K} \bar{p}_m(k) \log \bar{p}_m(k) \\ &= -(0.5 \log 0.5 + 0.3 \log 0.3 + 0.2 \log 0.2) \\ &= -(0.5 \times (-0.6931) + 0.3 \times (-1.204) + 0.2 \times (-1.6094)) \\ &= 1.03 \end{aligned}$$

1.5.4 Splitting Summary

For each iteration, we calculate the Loss (or Gain) for *all* features $j = 1, 2, \ldots, p$ and *all* possible split points. Choose the pair that minimizes the loss (or maximizes the gain):

$$\begin{aligned} (j^*, s^*) &= \arg\min_{j,s} \text{Loss}(j, s) \\ &= \arg\max_{j,s} \text{Gain}(j, s) \end{aligned}$$

where $\text{Loss}(j, s)$ is the loss after splitting the current node on the j predictor at split point s.

1.6 Stopping and Pruning

- **Tree Size**:
  - A large tree (i.e., many leaf nodes with few observations) risks **overfitting**, meaning the model captures noise in the training data rather than the underlying pattern.
  - A small tree may be too simple, failing to capture important structure, leading to **underfitting**.
  - Tree size is a **tuning parameter** that controls the model's complexity. The optimal tree size should be determined adaptively from the data, e.g. through cross-validation.
- **Early Stopping**:
  - Stop growing the tree when the improvement in the loss function becomes insignificant, similar to **forward stepwise selection**.
  - However, be cautious: a seemingly unimportant early split might enable better splits deeper in the tree (short-sightedness).
- **Pruning**:
  - Build a fully grown tree (allowing it to overfit with small terminal nodes), then **prune** back unnecessary branches to reduce overfitting, similar to **backward stepwise selection**.
  - Pruning removes splits that do not contribute significantly to reducing the training loss, making the tree more generalizable and reducing the variance.

1.6.1 Cost Complexity Pruning

- Let N_m be the number of observations in node R_m and $Q_m(T)$ represent the loss in region m for a given tree T. For example, using sum of squared errors as the loss function, we get

$$Q_m(T) = \sum_{\{i: x_i \in R_m\}} (y_i - \hat{c}_m)^2$$

- where $\hat{c}_m$ is the predicted value for observations within the region R_m. Typically, this is the mean of y_i for the observations in the region.

- *Weakest link pruning*: This method successively collapses/removes the internal node that produces the smallest increase in the total loss $\sum_{m=1}^{|T|} Q_m(T)$. This process produces a finite sequence of increasingly smaller sub-trees, each representing a pruned version of the original tree.
- For each sub-tree T, we define its *cost complexity*

$$\begin{aligned} C_\lambda(T) &= \frac{1}{n}\sum_{m=1}^{|T|} Q_m(T) + \lambda|T| \\ &= \text{Loss(T)} + \lambda\ \text{Penalty(T)} \end{aligned}$$

  where λ is a tuning parameter that controls the overall complexity of the tree.
  - Note: The *complexity* of a tree in this setting is measured by the *number of leaf nodes*, $|T|$

1.6.2 Penalty Tuning

- For each λ, there is a unique smallest sub-tree T_λ that minimizes $C_\lambda(T)$.
- The sequence of sub-trees from weakest link pruning contains every T_λ
- The tuning parameter, λ can be chosen by: cross-validation, AIC/BIC, Out-of-Bag (OOB), etc.

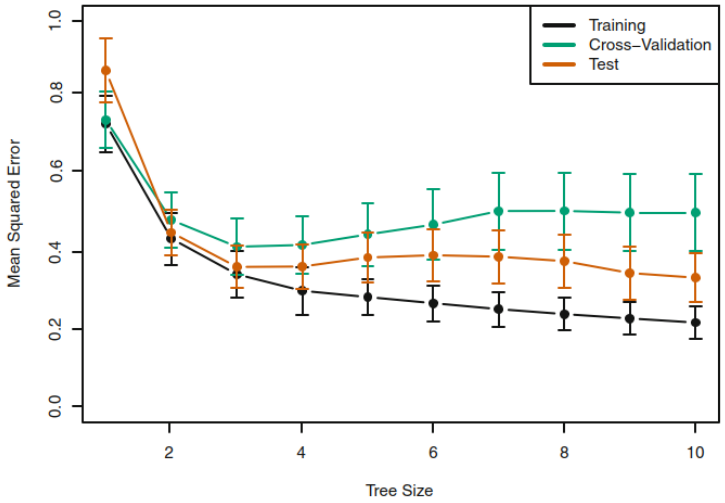


FIGURE 8.5. *Regression tree analysis for the* `Hitters` *data. The training, cross-validation, and test MSE are shown as a function of the number of terminal nodes in the pruned tree. Standard error bands are displayed. The minimum cross-validation error occurs at a tree size of three.*

1.7 Special Considerations

1.7.1 Missing Predictor Values

1. Omit observations with missing values.
   - This is the simplest approach but can lead to a loss of important predictive information, especially if missing values are not randomly distributed.
2. Create a new category for missing values (Categorical predictors)

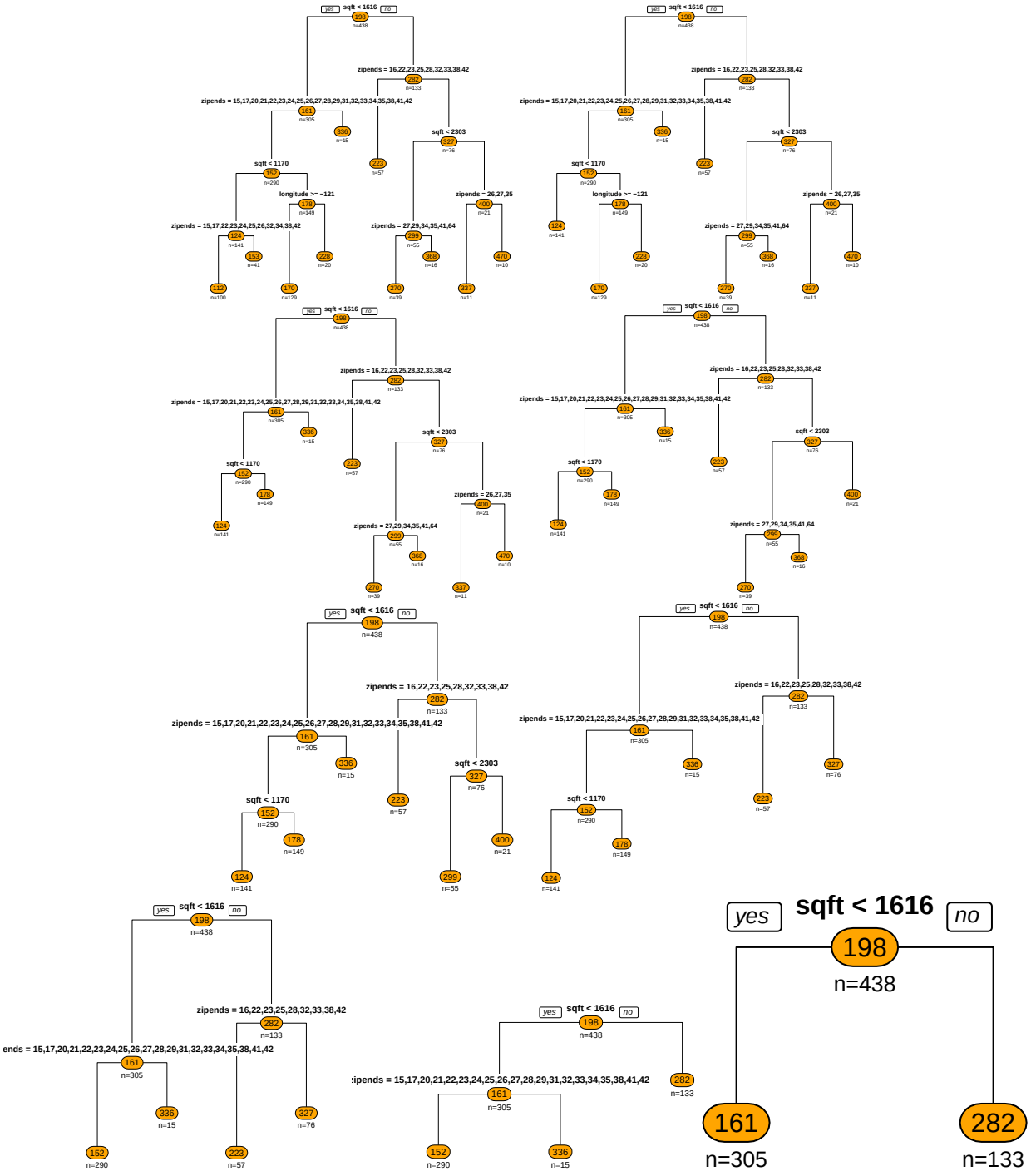


Figure 2: Weakest Link Pruning. Tree models housing prices in Sacramento, CA.

- For categorical predictors, create an additional level `"missing"`. This allows the tree to capture any potential patterns in the data where *missing* values may be systematically related to the outcome.

3. Surrogate splits
   - At every split in the tree, generate a list of *surrogate splits* that mimic the original split using other available predictors.
   - During prediction, if an observation has a missing value for the primary splitting variable, use the surrogate splits to determine which child node the observation should be directed to.
4. Imputation. Estimate missing values before splitting.
   a. **Global imputation:** before constructing the tree, replace all missing values. The imputation could be basic, like mean or median substitution, or more advanced multiple imputation.
   b. **Node Imputation:** perform imputation only using the data at each node (same branch of the tree). This approach uses the "nearest neighbors" observations to perform imputation instead of all available observations.
5. Random assignment
   - Another simple approach is to *randomly send* observations with missing values in the splitting variable to a child node.
   - A non-random approach is to send observations with missing values to the child node with the most other observations.

1.7.2 Binary Splitting

- Multiway splits are possible for some implementations (e.g., CHAID, C5.0). But a multiway split can partition the data too quickly and not lead to good subsequent splits.
- Multiway splits can still be achieved from binary splits trees using a combination of binary splits
  - I.e., split on X_1 at s_1 and then split again on X_1 at s_2
  - This will/should happen when the true response is not a constant.

1.7.3 Variable/Feature Importance

In prediction trees, several methods can be used to measure the *importance* of a feature. Feature importance helps identify which variables have the greatest influence on prediction. :

1. Frequency of feature used in splits
   - One simple way to measure the importance of a feature is to count how often it is used to make a split in the tree.
   $$\mathcal{I}_j(T) = \sum_t \mathbb{1}(\text{split } t \text{ uses feature} j)$$
   - Here, $\mathcal{I}_j(T)$ represents the importance of feature j in tree T, and the sum is over all splits t in tree T. The indicator function $\mathbb{1}$ returns 1 if split t uses feature j, and 0 otherwise.
   - This method provides a basic frequency count, but it doesn't consider the significance of the splits or how much they reduce the prediction error.
2. Predictive improvement of split
   - A more informative approach is to measure the total *reduction in loss* (or increase in gain) due to splits involving the feature. This method considers both the frequency of feature use and the effectiveness of the split.

- In CART (Classification and Regression Trees), the importance of a feature also includes its use in surrogate splits.

The importance of feature j in tree T can be expressed as:

$$\mathcal{I}_j(T) = \sum_t \text{gain}(t) \cdot \mathbb{1}(\text{split } t \text{ uses feature} j)$$

- In this equation, the importance of feature j is the total *gain* across all splits that involve feature j. Gain refers to the reduction in the chosen loss function (e.g., Gini index or mean squared error) for each split. - This method is a *weighted* version of the previous approach, giving more emphasis to features that contribute to the reduction in error.

3. Permutation-Based Importance (Prediction Version)
   - Another popular method for assessing feature importance is *permutation-based importance*. This method evaluates how much the predictive performance of the model decreases when the values of a feature are permuted (shuffled).
   - The process is as follows:
     1. First, calculate the tree's performance on a hold-out validation set.
     2. Then permute, shuffle, or resample the values of feature j in the validation set.
     3. Reassess the model's performance with the permuted/shuffled/resampled feature.
   - The importance of feature j is measured by the decrease in performance due to the permutation:

$$\mathcal{I}_j(T) = \text{Loss(using permuted feature } j) - \text{Loss(original)}$$

   - A large increase in the loss (i.e., worse model performance) after permutation indicates that the feature is influential, as the model relied heavily on it to make good predictions.
   - This method is particularly useful as it considers the *global impact* of the feature on the model's predictive ability, not just its role in specific splits.

Note

This is only one type of permutation importance. We will explore other ways to assess variable importance later in the course.

1.8 Prediction Tree Advantages

- Handles both categorical and continuous data consistently
  - Trees will work with both categorical and continuous predictors, without requiring extensive preprocessing or transformations.
- Automatic variable selection
  - Trees automatically perform variable selection by choosing only the most important predictors for splitting at each node. Any predictor not used in a split is effectively excluded from the model without explicit feature elimination.
- Automatically discovers interactions between multiple predictors
  - Trees inherently capture interactions between predictors. The depth of the tree governs the complexity of these interactions, allowing for multiple levels of interaction to be modeled without requiring manual specification.
- Locally adaptive estimates
  - Because the tree partitions the feature space based on the observed data, it provides *locally adaptive* estimates. This means that predictions will be based on the outcomes from the similar observations.

- Invariant to monotonic transformations
  - Trees are typically invariant to monotone transformations of the predictors (e.g., log transformations, scaling). This property holds because trees rely on the relative ordering of feature values rather than their absolute values.
- Robust to outliers in feature space
  - Trees can be robust to outliers in feature space. This robustness arises because trees split based on the relative ordering of feature values rather than the raw feature values themselves. Trees also fit a constant in each region. Thus, outliers in feature space are less likely to dominate the model compared to other techniques, like linear regression.
- Easy to interpret / Transparent
  - One of the key strengths of prediction trees is their interpretability. The hierarchical structure of a tree, where each split is a simple decision rule, makes it easy for users to understand and explain the model's predictions.

1.9 Tree Limitations

- Instability (high variance) due to the greedy hierarchical structure.
  - Trees are prone to instability (i.e., high variance). A small change in the data, particularly at the top split, can lead to a significantly different tree structure, causing a cascading effect throughout the tree. This makes trees highly sensitive to small variations in the dataset.
  - Ensemble methods like bagging (e.g., random forests) and boosting can help mitigate this issue by aggregating multiple trees to reduce variance.
- Difficulty capturing additive structure.
  - Trees will struggle to model additive relationships between predictors. For instance, if the true relationship is a linear combination of features (e.g., $y = \beta_1 x_1 + \beta_2 x_2$), trees may fail to recognize this pattern efficiently.
  - Trees are better suited for detecting interactions and non-linear relationships, but they may perform poorly when the data have a strong additive structure, unless ensemble methods are used.

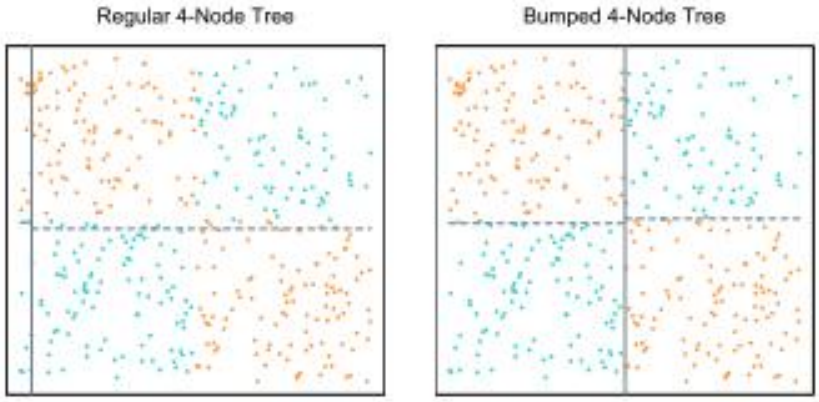


FIGURE 8.13. *Data with two features and two classes (blue and orange), displaying a pure interaction. The left panel shows the partition found by three splits of a standard, greedy, tree-growing algorithm. The vertical grey line near the left edge is the first split, and the broken lines are the two subsequent splits. The algorithm has no idea where to make a good initial split, and makes a poor choice. The right panel shows the near-optimal splits found by bumping the tree-growing algorithm 20 times.*

- Bias towards dominant features.
  - Trees can be biased towards categorical features that have many levels. This can result in the tree splitting on these features, even if they are not the most informative.
- Lack of smooth predictions.
  - Remember that trees generate piecewise constant prediction surfaces that can result in abrupt jumps between predictions for neighboring observations.
  - Because all observations that fall into the same leaf node get the same prediction, there can be multiple observations with the same predicted values. Ensure that subsequent evaluation metrics can properly handle ties.
- Predictive Bias (or mis-calibration)
  - tree will often produce predictions that are biased (also known as mis-calibrated). We will study this topic later in the semester.

1.10 Prediction Trees in R

Main R packages: `tree` and `rpart` and `party`

2 Trees Demo

2.1 Required R Packages

```
library(ISLR)          # Hitters baseball data
library(rpart)         # classification and regression trees (CART)
library(rpart.plot)    # for `prp()` which allows more plotting control for trees
library(randomForest)  # for `randomForest()` function
library(tidyverse)     # data manipulation and visualization
```

2.2 Baseball Salary Data

The goal is to build models to predict the salary of baseball players

```
#-- Make Baseball Data
#   Goal is to predict the log Salary

library(ISLR)
Hitters = ISLR::Hitters %>%
  filter(!is.na(Salary)) %>%          # remove missing Salary
  # mutate(Salary = log(Salary)) %>%   # convert to log Salary
  rename(Y = Salary)

set.seed(2019) # choose 200 samples for training (leaving only 63 for testing)
train.ind = sample(nrow(Hitters), size=200)
bball = Hitters[train.ind, ]

#- test data
X.test = Hitters[-train.ind, ] %>% select(-Y)
Y.test = Hitters[-train.ind, ] %>% pull(Y)
```

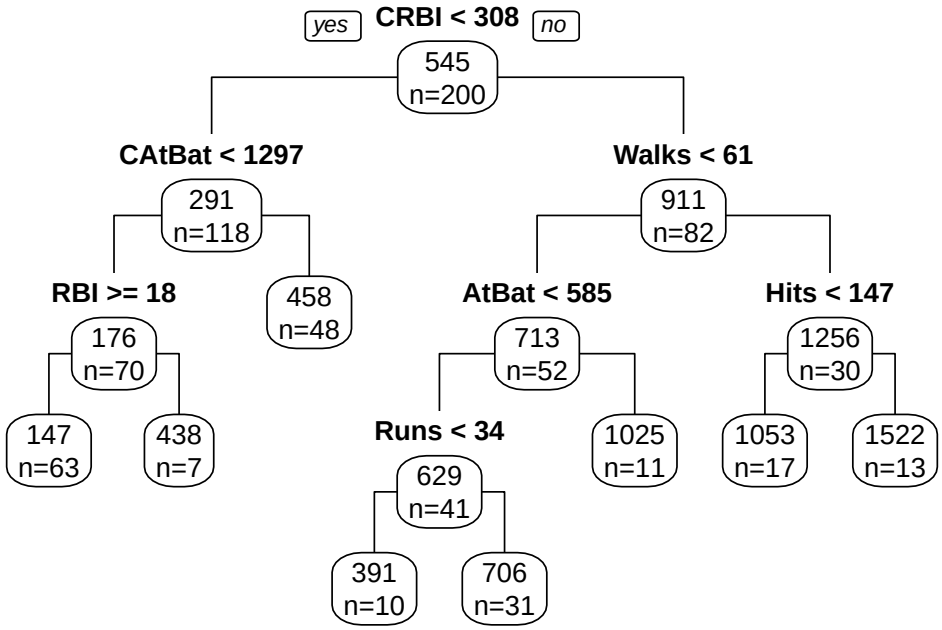
```
bball %>% arrange(-Y) %>% head() %>% as_tibble(rownames = "name")
#> # A tibble: 6 x 21
#>   name       AtBat  Hits HmRun  Runs   RBI Walks Years CAtBat CHits CHmRun CRuns
#>   <chr>      <int> <int> <int> <int> <int> <int> <int>  <int> <int>  <int> <int>
#> 1 -Eddie Mu~   495   151    17    61    84    78    10   5624  1679    275   884
#> 2 -Jim Rice    618   200    20    98   110    62    13   7127  2163    351  1104
#> 3 -Mike Sch~    20     1     0     0     0     0     2     41     9      2     6
#> 4 -Don Matt~   677   238    31   117   113    53     5   2223   737     93   349
#> 5 -Ozzie Sm~   514   144     0    67    54    79     9   4739  1169     13   583
#> 6 -Gary Car~   490   125    24    81   105    62    13   6063  1646    271   847
#> # i 9 more variables: CRBI <int>, CWalks <int>, League <fct>, Division <fct>,
#> #   PutOuts <int>, Assists <int>, Errors <int>, Y <dbl>, NewLeague <fct>
```

2.3 Regression Tree

2.3.1 Build Tree

```
#################################################################
#-- Regression Trees in R
# trees are in many packages: rpart, tree, party, ...
# there are also many packages to display tree results
#
# Formulas: you don't need to specify interactions as the tree does this
#  naturally.
#################################################################
#-- Build Tree
```

```r
library(rpart)
tree = rpart(Y~., data=bball)
summary(tree, cp=1)
#> Call:
#> rpart(formula = Y ~ ., data = bball)
#>   n= 200
#>
#>         CP nsplit rel error xerror   xstd
#> 1 0.39734      0    1.0000 1.0071 0.1517
#> 2 0.11976      1    0.6027 0.6265 0.1140
#> 3 0.04832      2    0.4829 0.5959 0.1127
#> 4 0.03453      3    0.4346 0.5672 0.1154
#> 5 0.02898      4    0.4001 0.6263 0.1211
#> 6 0.01593      5    0.3711 0.6296 0.1208
#> 7 0.01143      6    0.3551 0.6272 0.1207
#> 8 0.01000      7    0.3437 0.6294 0.1204
#>
#> Variable importance
#>   CRBI  CRuns  CHits CAtBat CWalks CHmRun  Walks   Runs   Hits  AtBat    RBI
#>     16     15     14     14     14     12      5      4      2      2      1
#>  HmRun
#>      1
#>
#> Node number 1: 200 observations
#>   mean=545.1, MSE=2.347e+05
length(unique(tree$where))        # number of leaf nodes
#> [1] 8

#-- Plot Tree
library(rpart.plot)   # for prp() which allows more plotting control
prp(tree, type=1, extra=1, branch=1)

# rpart() functions can also plot (just not as good):
#   plot(tree, uniform=TRUE)
#   text(tree, use.n=TRUE, xpd=TRUE)
```



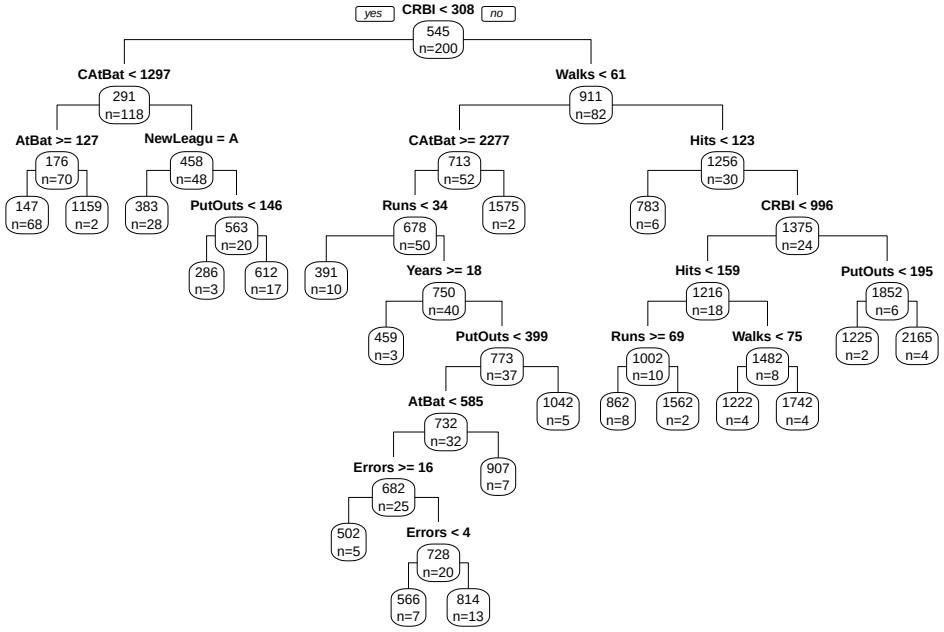
2.3.2 Evaluate Tree

```
#- mean squared error function
mse <- function(yhat, y){
  yhat = as.matrix(yhat)
  apply(yhat, 2, \(f) mean((y-f)^2))
}


mse(predict(tree), bball$Y)            # training error
#> [1] 80680
mse(predict(tree, X.test), Y.test)     # testing error
#> [1] 59872
```

Build a more complex tree

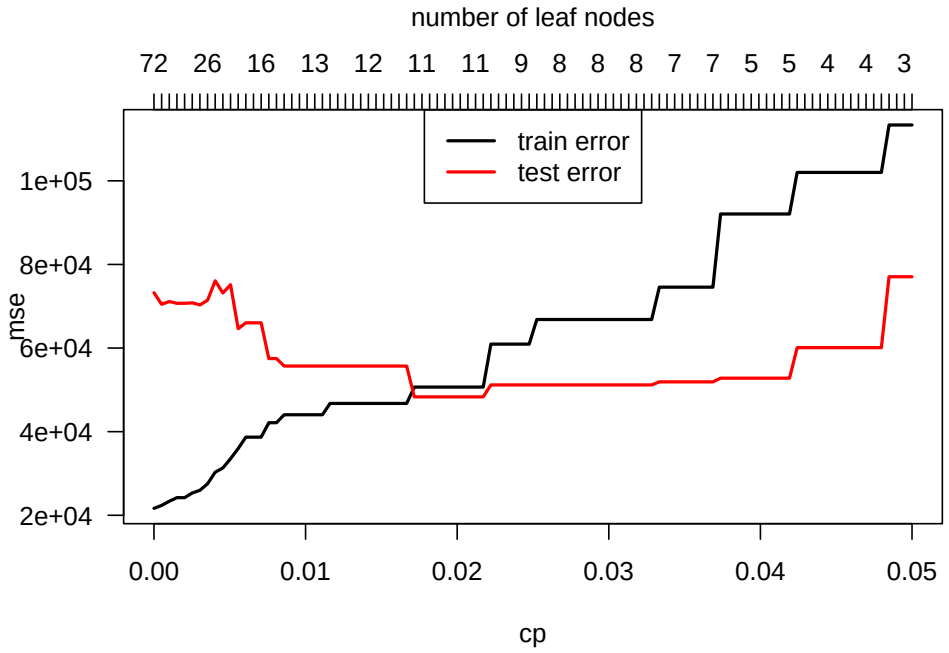
```
#-- More complex tree
# see ?rpart.control() for details
# xval: number of cross-validations
# minsplit: min obs to still allow a split
# cp: complexity parameter

tree2 = rpart(Y~., data=bball, xval=0, minsplit=5, cp=0.005)
summary(tree2, cp=1)
#> Call:
#> rpart(formula = Y ~ ., data = bball, xval = 0, minsplit = 5,
#>     cp = 0.005)
#>   n= 200
#>
#>           CP nsplit rel error
#> 1  0.397337      0    1.0000
#> 2  0.119759      1    0.6027
#> 3  0.048320      2    0.4829
#> 4  0.042372      3    0.4346
#> 5  0.037284      4    0.3922
#> 6  0.032953      6    0.3176
#> 7  0.025089      7    0.2847
#> 8  0.021944      8    0.2596
#> 9  0.021814      9    0.2377
#> 10 0.016670     10    0.2158
#> 11 0.011547     11    0.1992
#> 12 0.008095     12    0.1876
#> 13 0.007354     13    0.1795
#> 14 0.005854     15    0.1648
#> 15 0.005786     16    0.1590
#> 16 0.005148     17    0.1532
#> 17 0.005000     19    0.1429
#>
#> Variable importance
#>   CRuns    CRBI  CAtBat   CHits  CWalks  CHmRun   Walks    Runs    Hits   AtBat
#>      14      14      13      12      11      10       6       5       5       4
#> PutOuts     RBI   Years Assists   HmRun  Errors
#>       1       1       1       1       1       1
#>
#> Node number 1: 200 observations
#>   mean=545.1, MSE=2.347e+05
length(unique(tree2$where))
#> [1] 20
```

```r
prp(tree2, type=1, extra=1, branch=1)

mse(predict(tree2), bball$Y)              # training error
#> [1] 33541
mse(predict(tree2, X.test), Y.test)       # testing error
#> [1] 75146
```



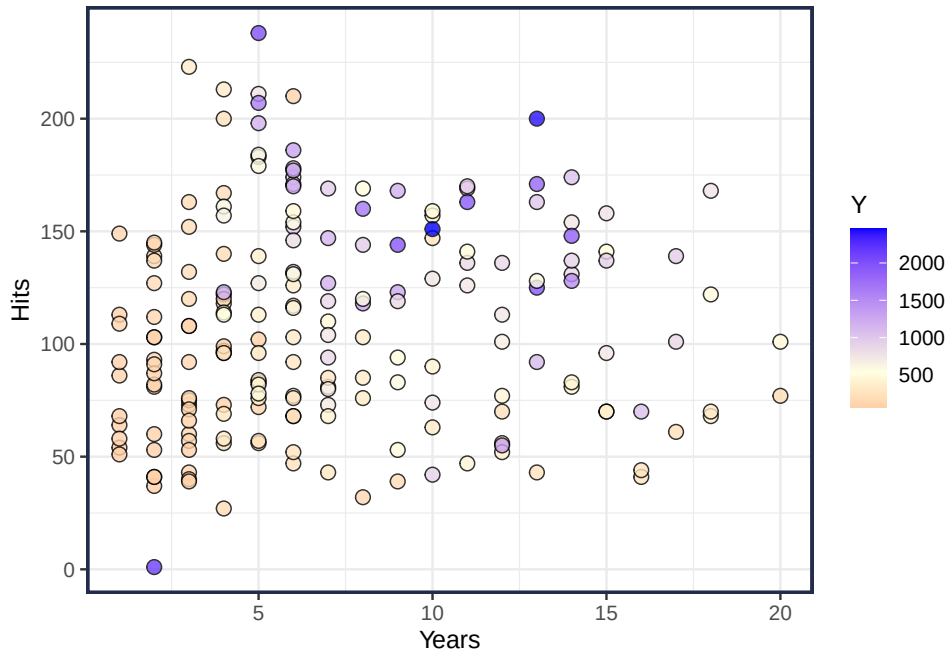
Now, fit a set of trees for sequence of `cp` values.

```r
cp = seq(.05,0,length=100)  # cp is like a penalty on the tree size
for(i in 1:length(cp)){
  if(i == 1){train.error = test.error = nleafs = numeric(length(cp))}
  tree.fit = rpart(Y~.,data=bball, xval=0, minsplit=5, cp=cp[i])
  train.error[i] = mse(predict(tree.fit),bball$Y)            # training error
  test.error[i] = mse(predict(tree.fit,X.test),Y.test)    # testing error
  nleafs[i] = length(unique(tree.fit$where))
}

plot(range(cp),range(train.error,test.error),typ='n',xlab="cp",ylab="mse",las=1)
lines(cp,train.error,col="black",lwd=2)
lines(cp,test.error,col="red",lwd=2)
legend("top",c('train error','test error'),col=c("black","red"),lwd=2)
axis(3,at=cp,labels=nleafs)
mtext("number of leaf nodes",3,line=2.5)
```

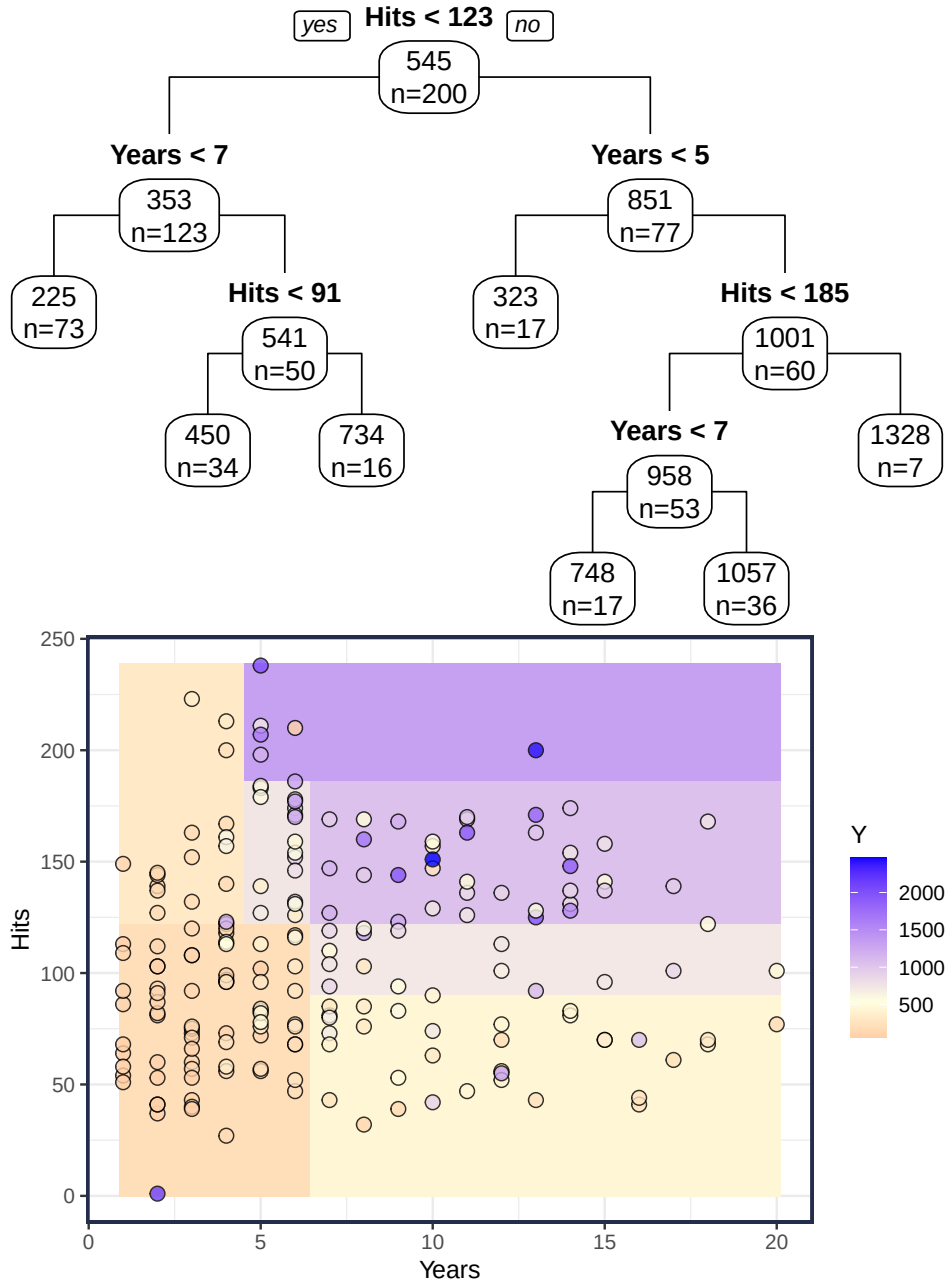


2.3.3 Regression Tree example with 2 dimensions only

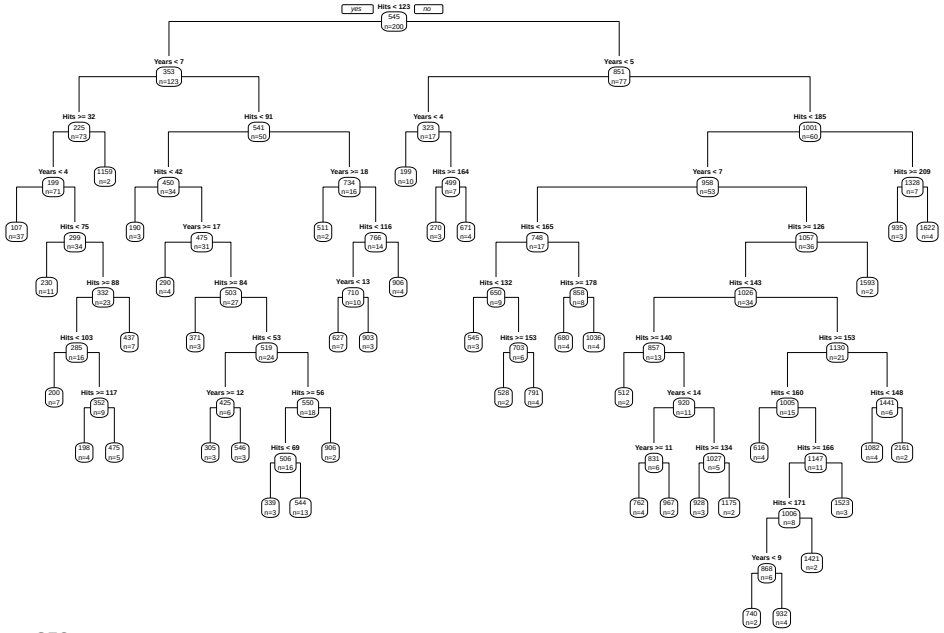
Consider the two variables `Years` and `Hits` and their relationship to `Y`.

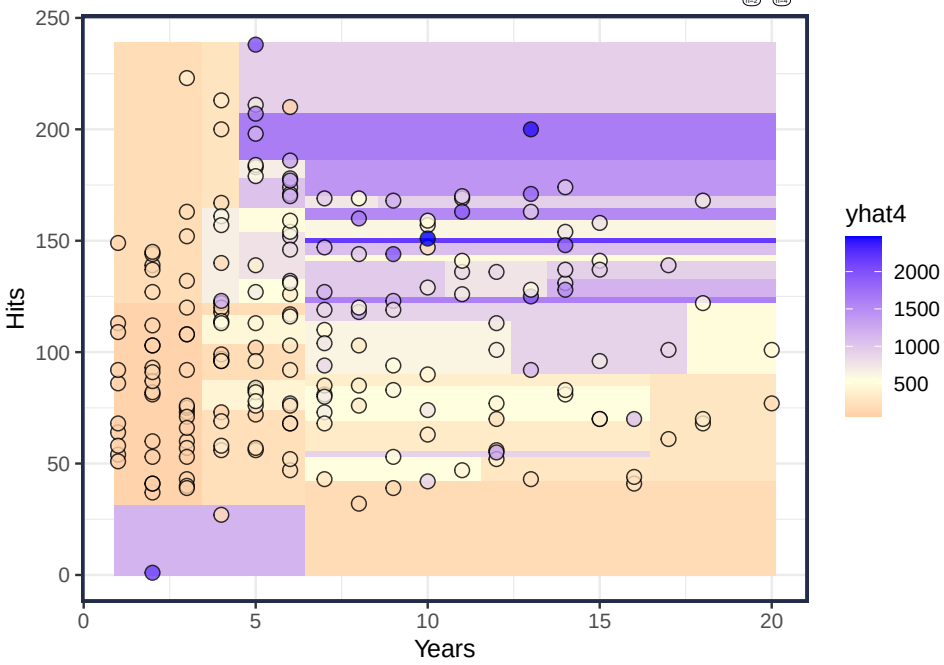


Let's fit a tree with the two predictors



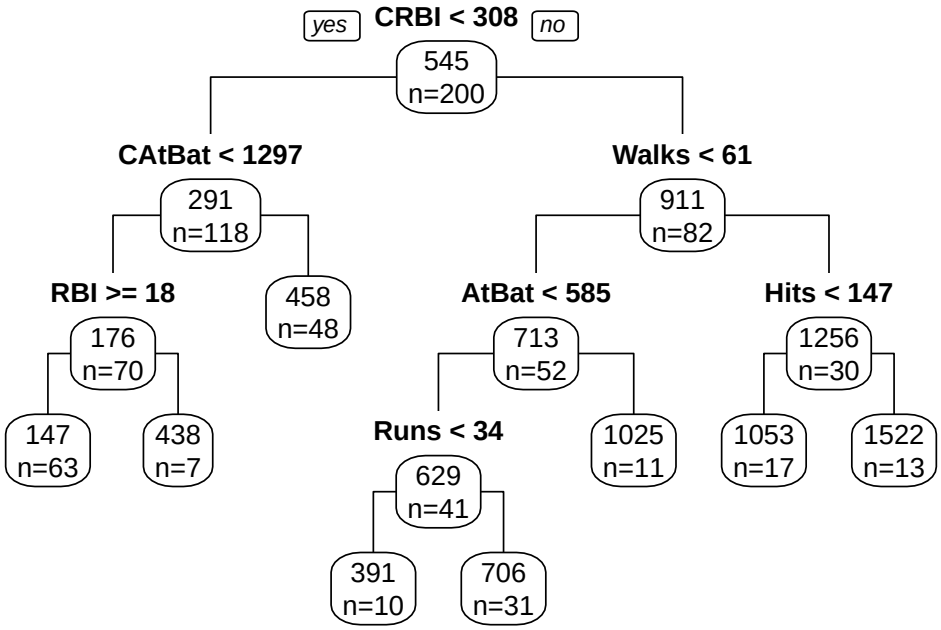
And we can also use more complex trees:


yes
Hits < 123
no
545
n=208
Years < 7
353
n=128
Years < 5
851
n=77
Hits >= 32
325
n=73
Hits < 91
541
n=50
Years < 4
323
n=17
Hits < 185
1001
n=60
Years < 4
199
n=71
1159
n=2
Hits < 42
450
n=34
Years >= 18
734
n=16
199
n=10
Hits >= 164
495
n=7
Years < 7
958
n=53
Hits >= 209
1328
n=7
107
n=37
Hits < 75
299
n=34
190
n=3
Years >= 17
475
n=31
511
n=2
Hits < 116
766
n=14
270
n=3
671
n=4
Hits < 165
748
n=17
Hits >= 126
1057
n=36
935
n=3
1622
n=4
230
n=11
Hits >= 88
332
n=23
290
n=4
Hits >= 84
503
n=27
Years < 13
710
n=15
906
n=4
Hits < 132
658
n=9
Hits >= 178
658
n=8
Hits >= 143
1028
n=33
1593
n=2
Hits < 103
385
n=16
437
n=7
371
n=3
Hits < 53
518
n=24
627
n=7
903
n=3
545
n=3
Hits >= 153
793
n=6
680
n=4
1036
n=4
Hits >= 140
857
n=13
Hits >= 153
1139
n=21
200
n=7
Hits >= 117
332
n=9
Years >= 12
428
n=6
Hits >= 56
550
n=16
528
n=2
791
n=4
512
n=2
Years < 14
930
n=11
Hits < 160
1065
n=15
Hits < 148
1141
n=6
198
n=4
475
n=5
305
n=3
548
n=3
Hits < 69
906
n=16
906
n=2
Years >= 11
831
n=6
Hits >= 134
1027
n=5
616
n=4
Hits >= 166
1147
n=11
1092
n=4
2161
n=2
539
n=3
544
n=13
762
n=4
867
n=2
928
n=3
1175
n=2
Hits < 171
1006
n=8
1523
n=3
Years < 9
868
n=6
1421
n=2
740
n=2
992
n=4


250
200
150
100
50
0
Hits
0
5
10
15
20
Years
yhat4
2000
1500
1000
500

2.4 Details of Splitting (for Regression Trees)

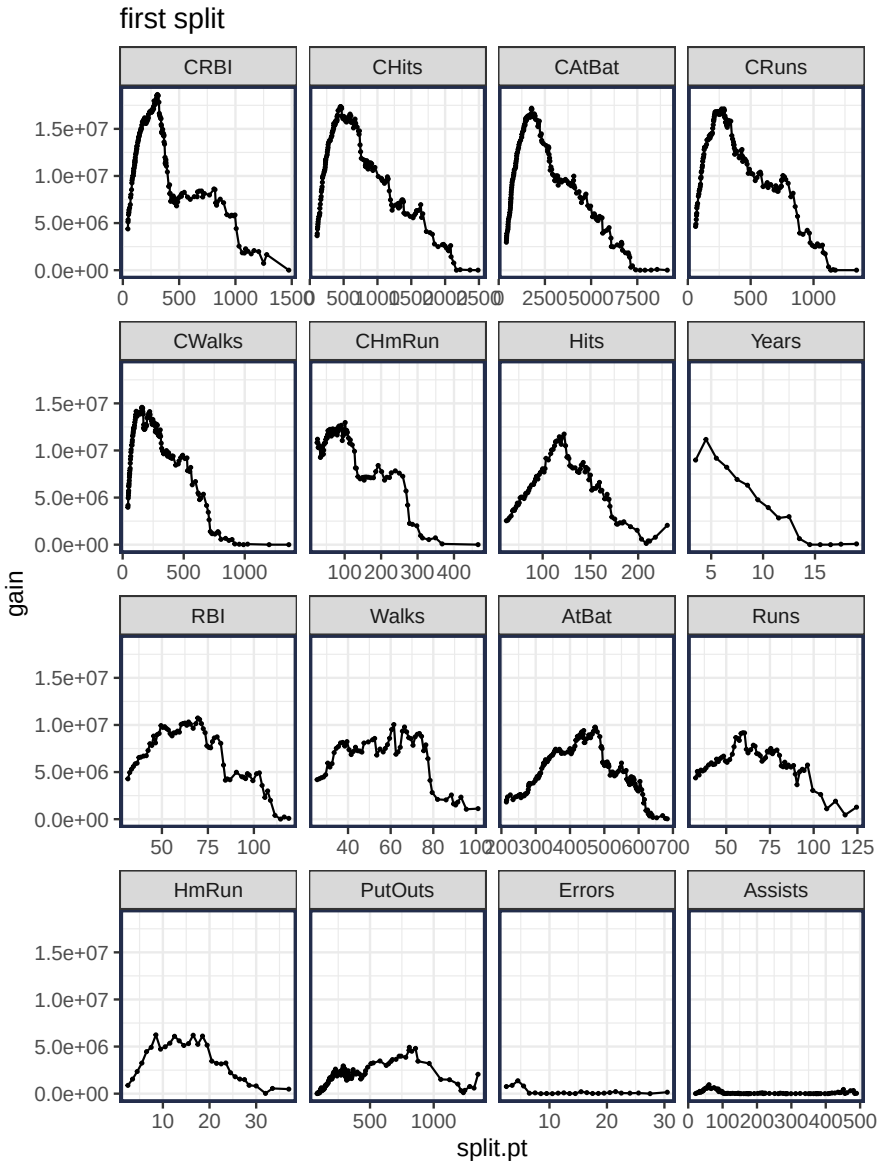
Going back to building a tree with all predictor variables. Recall that `CRBI` is split at `307.5`.



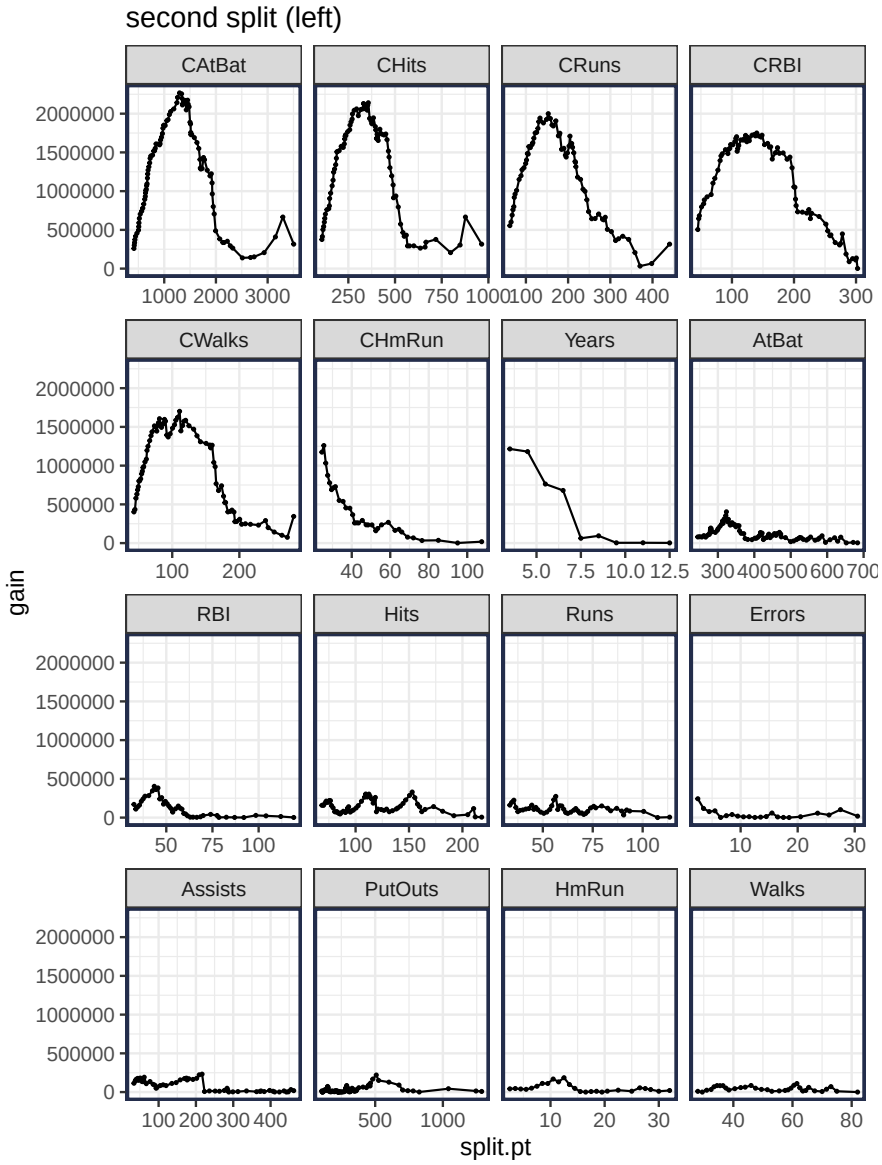
2.4.1 First Split

Under the hood, trees will search all possibly split points for all predictor variables. It will use the variable and associated split point that has the maximum gain (or improvement in evaluation metric).

var	split.pt	n.L	n.R	est.L	est.R	SSE.L	SSE.R	SSE	gain
CRBI	307.5	118	82	290.5	911.5	8010622	20282448	28293070	18653665
CHits	457.5	93	107	229.0	819.9	5299313	24273274	29572587	17374148
CAtBat	1779.5	95	105	236.8	824.0	5574003	24175441	29749444	17197291
CRuns	288.0	109	91	277.7	865.4	7452616	22368698	29821314	17125421
CWalks	157.5	90	110	246.6	789.3	6768757	25600645	32369403	14577333
CHmRun	101.5	156	44	409.9	1024.6	21520628	12459474	33980101	12966634
Hits	122.5	123	77	353.4	851.4	11979104	23225406	35204510	11742225
Years	4.5	69	131	219.5	716.6	5807172	29967467	35774639	11172096
RBI	69.5	147	53	406.0	931.1	18079731	18126052	36205783	10740952
Walks	61.5	155	45	424.4	961.0	19360871	17541643	36902514	10044222
AtBat	473.5	124	76	372.1	827.4	13829528	23351912	37181440	9765295
Runs	59.5	115	85	361.0	794.2	11945002	25829983	37774985	9171750
HmRun	8.5	94	106	357.6	711.4	12526389	28182028	40708417	6238318
PutOuts	809.0	186	14	502.1	1116.7	35199872	6828501	42028373	4918362
Errors	4.5	71	129	433.5	606.5	7733767	37842076	45575843	1370892
Assists	60.5	111	89	483.4	622.1	20984841	25011176	45996017	950719

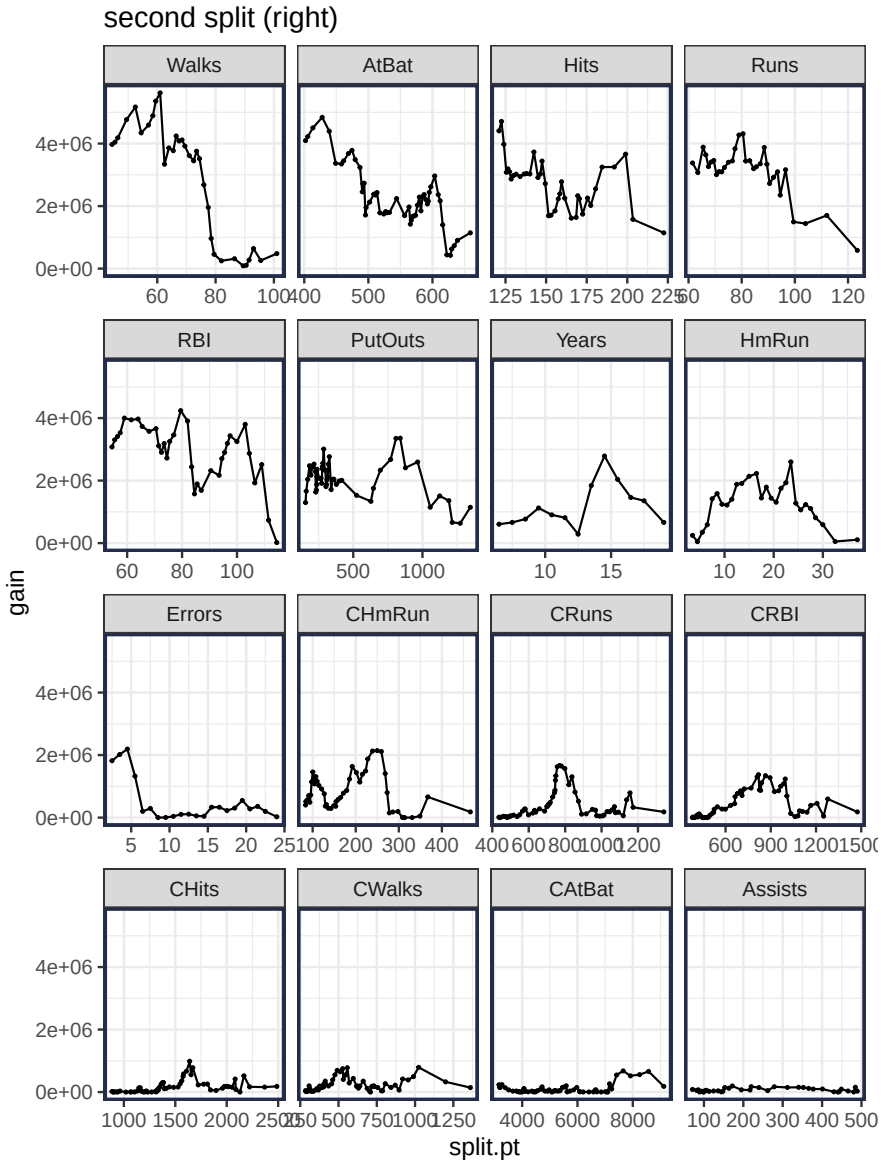

first split
CRBI
CHits
CAtBat
CRuns
CWalks
CHmRun
Hits
Years
RBI
Walks
AtBat
Runs
HmRun
PutOuts
Errors
Assists
gain
split.pt
1.5e+07
1.0e+07
5.0e+06
0.0e+00

2.4.2 Second Split (left): CAtBat < 1296.5



var	split.pt	n.L	n.R	est.L	est.R	SSE.L	SSE.R	SSE	gain
CAtBat	1296.5	70	48	175.7	458.0	4142792	1599385	5742177	2268445
CHits	357.0	77	41	192.2	475.1	4385523	1484232	5869754	2140868
CRuns	153.0	68	50	178.9	442.3	4202620	1808176	6010797	1999825
CRBI	140.0	73	45	194.9	445.6	4612088	1648340	6260429	1750193
CWalks	111.0	74	44	197.9	446.2	4682658	1626914	6309572	1701050
CHmRun	25.5	73	45	209.4	422.1	4950482	1800792	6751274	1259349
Years	3.5	48	70	168.0	374.6	4065840	2729314	6795153	1215469
AtBat	323.5	50	68	222.3	340.7	4391102	3215583	7606685	403937
RBI	43.5	64	54	236.8	354.2	5514802	2092255	7607057	403565
Hits	153.0	105	13	271.9	440.8	7158659	522192	7680851	329771
Runs	56.5	77	41	255.4	356.5	5887826	1849136	7736962	273661
Errors	2.5	17	101	179.9	309.2	172699	7594684	7767383	243239
Assists	216.5	90	28	265.8	369.9	3189715	4589431	7779145	231477
PutOuts	508.0	107	11	276.7	425.3	7138760	651614	7790374	220248
HmRun	12.5	84	34	265.3	352.8	6395242	1429943	7825185	185437
Walks	61.5	103	15	278.9	370.7	7018267	881993	7900261	110361

2.4.3 Second Split (right): Walks < 61



var	split.pt	n.L	n.R	est.L	est.R	SSE.L	SSE.R	SSE	gain
Walks	61.0	52	30	712.6	1256.2	5842326	8817840	14660166	5622282
AtBat	428.0	26	56	554.9	1077.0	2032940	13410065	15443005	4839443
Hits	122.5	30	52	595.9	1093.5	2674697	12898042	15572738	4709710
Runs	80.5	58	24	763.9	1268.1	9627677	6339423	15967100	4315348
RBI	79.5	57	25	760.9	1254.7	8028152	8016141	16044292	4238156
PutOuts	839.5	73	9	840.4	1487.7	14312345	2613789	16926134	3356314
Years	14.5	64	18	1009.2	563.9	16198744	1298244	17496988	2785460
HmRun	23.5	65	17	820.4	1259.5	14245529	3438753	17684283	2598166
Errors	4.5	29	53	690.4	1032.4	3230563	14859766	18090329	2192119
CHmRun	250.0	69	13	841.3	1283.9	11497937	6641903	18139840	2142609
CRuns	769.0	56	26	814.3	1120.7	8769462	9847003	18616465	1665983
CRBI	818.5	59	23	830.7	1118.7	9605277	9304706	18909983	1372466
CHits	1640.0	62	20	849.2	1104.6	10823636	8472147	19295784	986665
CWalks	1023.5	79	3	930.6	407.5	19425851	65712	19491563	790885
CAtBat	7653.5	77	5	934.6	554.8	19376696	228517	19605214	677234
Assists	172.0	60	22	941.0	830.9	16318132	3769004	20087137	195311

3 Bagging Trees

3.1 Better Trees

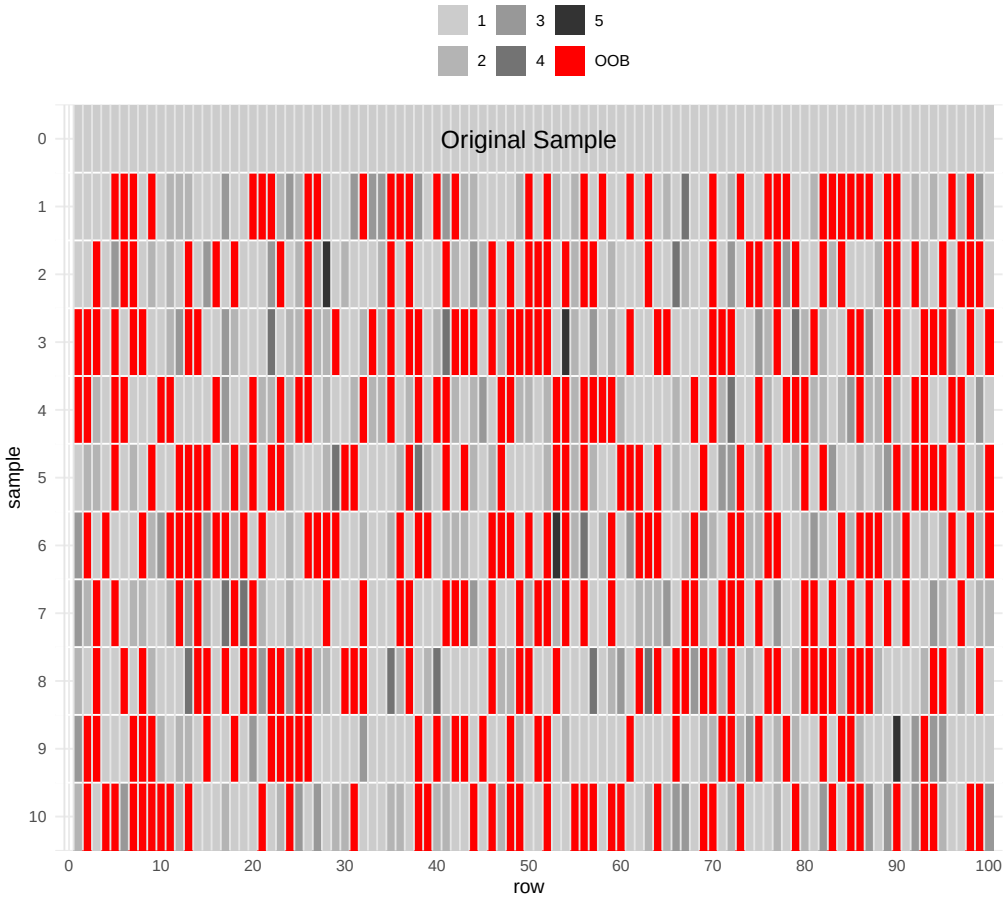
Due to the inherent instability of prediction trees, they are ideal candidates for methods that can reduce variance, such as **bagging** (Bootstrap Aggregating). Bagging works by generating multiple trees from bootstrap samples of the training data and averaging their predictions.

- Grow a set of B trees, each from a different bootstrap sample, and then average their predictions:

$$\hat{f}(x) = \frac{1}{B}\sum_{b=1}^{B} T(x;\hat{\theta}_b)$$

  - $\hat{\theta}_b$ represents the parameters for tree b, including the split variables, cutpoints, and terminal node values, all derived from the bootstrap sample $\mathcal{D}_b$.
  - This averaging over many trees reduces the sensitivity of the model to small changes in the data, addressing the instability and high variance of single trees.
- Bagging = Bootstrap Aggregating
  - The term "bagging" is a contraction of **B**ootstrap **Agg**regat**ing**.
- For a deeper dive into the methodology, see Breiman's article "Bagging Predictors" (1996, *Machine Learning*). This seminal paper offers detailed advice on when bagging is most beneficial and when it may not provide significant improvements.
  - For example, Bagging works well in reducing the **variance** of models but does not significantly reduce **bias**. Therefore, bagging is most effective for models prone to high variance, such as **deep** prediction trees.
- Bagging produces an ensemble model:
  - By combining multiple trees, bagging creates an *ensemble model* that averages the predictions of individual trees, smoothing out noise and providing more stable predictions.
- Aggregation of Bagged Predictors:
  a. *regression*: the final prediction is the average of the predictions from the individual tres.

b. *classification*: the final predicted probabilities are the average of the probability estimates from the individual trees. Another option is to use *majority voting* where each tree votes for the class label, but take care to adjust for class imbalance or unequal misclassification costs.



3.1.1 Variance Reduction with Bagging

Note

A helpful probability cheatsheet can be found here: https://github.com/wzchen/probability_cheatsheet/blob/master/probability_cheatsheet.pdf

Properties of Variance/Covariance

$$\begin{aligned}
\mathrm{V}(X) &= \mathrm{E}(X^2) - (\mathrm{E}(X))^2 \\
&= \mathrm{Cov}(X, X) \\
\mathrm{Cov}(X_1, X_2) &= \mathrm{E}(X_1 X_2) - \mathrm{E}(X_1)\,\mathrm{E}(X_2) \\
\mathrm{Cor}(X_i, X_j) &= \frac{\mathrm{Cov}(X_i, X_j)}{\sqrt{\mathrm{V}(X_i)\,\mathrm{V}(X_j)}} \\
\mathrm{V}(X_1 + X_2) &= \mathrm{V}(X_1) + \mathrm{V}(X_2) + 2\,\mathrm{Cov}(X_1, X_2) \\
\mathrm{V}\left(a \sum_{i=1}^{p} X_i\right) &= a^2 \sum_{i=1}^{p} \mathrm{V}(X_i) + 2a^2 \sum_{i<j} \mathrm{Cov}(X_i, X_j)
\end{aligned}$$

Variance Reduction

- Let θ be something we want to estimate (e.g., $\theta = f(x)$) and $\hat{\theta}$ an estimate.
- Suppose we have M models to estimate θ which produces the estimates $\{\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_M\}$
- One way to make an *ensemble prediction* is from the average

$$\bar{\theta} = \frac{1}{M} \sum_{i=1}^{M} \hat{\theta}_i$$

- The **expected value** of the ensemble is:

$$\mathrm{E}(\bar{\theta}) = \frac{1}{M} \sum_{i=1}^{M} \mathrm{E}(\hat{\theta}_i)$$

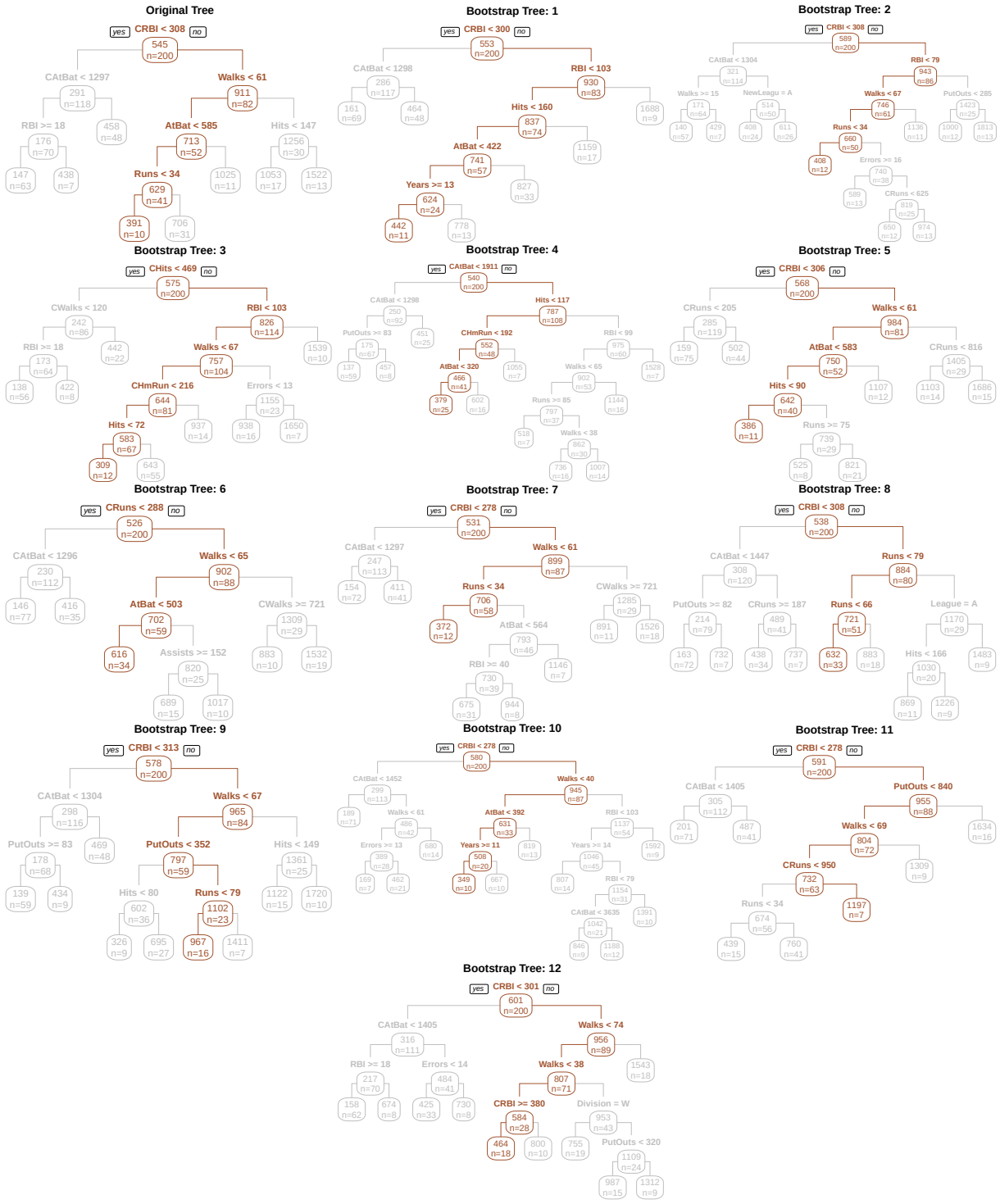
- The **variance** of the ensemble is:

$$\begin{aligned}
\mathrm{V}(\bar{\theta}) &= \frac{1}{M^2} \sum_{i=1}^{M} \mathrm{V}(\hat{\theta}_i) + \frac{2}{M^2} \sum_{i<j} \mathrm{Cov}(\hat{\theta}_i, \hat{\theta}_j) \\
&= \frac{1}{M^2} \sum_{i=1}^{M} \mathrm{V}(\hat{\theta}_i) + \frac{2}{M^2} \sum_{i<j} \sqrt{\mathrm{V}(\hat{\theta}_i)\,\mathrm{V}(\hat{\theta}_j)}\,\mathrm{Cor}(\hat{\theta}_i, \hat{\theta}_j)
\end{aligned}$$

- Thus to reduce the variance, we want to **use models that have low correlation**.
  - If $\mathrm{Cor}(\hat{\theta}_i, \hat{\theta}_j) = 0 \quad \forall i, j$, then variance is minimized (for example, when the models are *independent*)
  - If $\mathrm{Cor}(\hat{\theta}_i, \hat{\theta}_j) = 1 \quad \forall i, j$, then there is no (variance reduction) benefit of using an ensemble.
  - In Bagging, each *model* is a tree fit with a bootstrap sample.
  - For unstable models, like trees, the bagged models will have low correlation, but for more stable models, like linear regression, the bagged models will maintain high correlation.

3.2 Bagging Trees

Highlight node is for Pete Rose.



(ESL pg 587) "The essential idea in bagging is to average many noisy but approximately unbiased models, and hence reduce the variance."

- Thus when Bagging trees, **grow deep trees to reduce bias** and use **many bootstrap samples to**

reduce variance.

3.2.1 Correlation

These are the pairwise correlation between predictions from the trees.

3.2.2 Bagging can smooth predictions

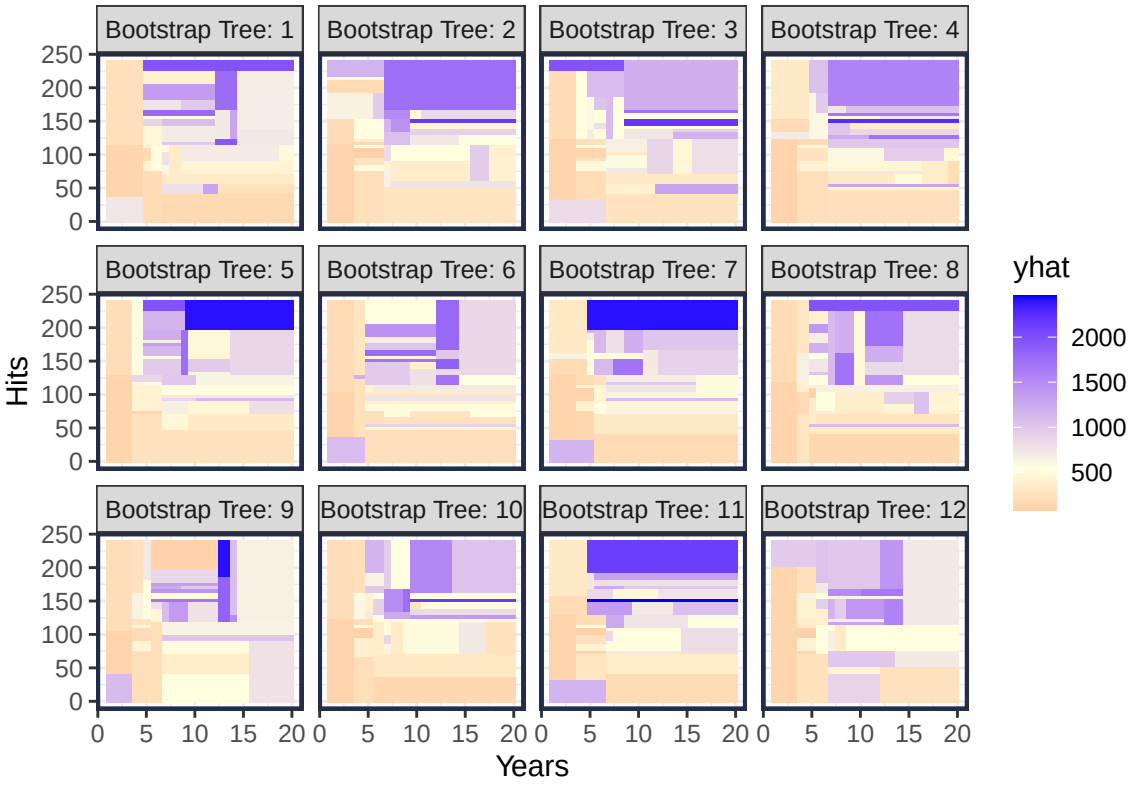


Figure 3: Bagging in 2 dimensions (Years and Hits)

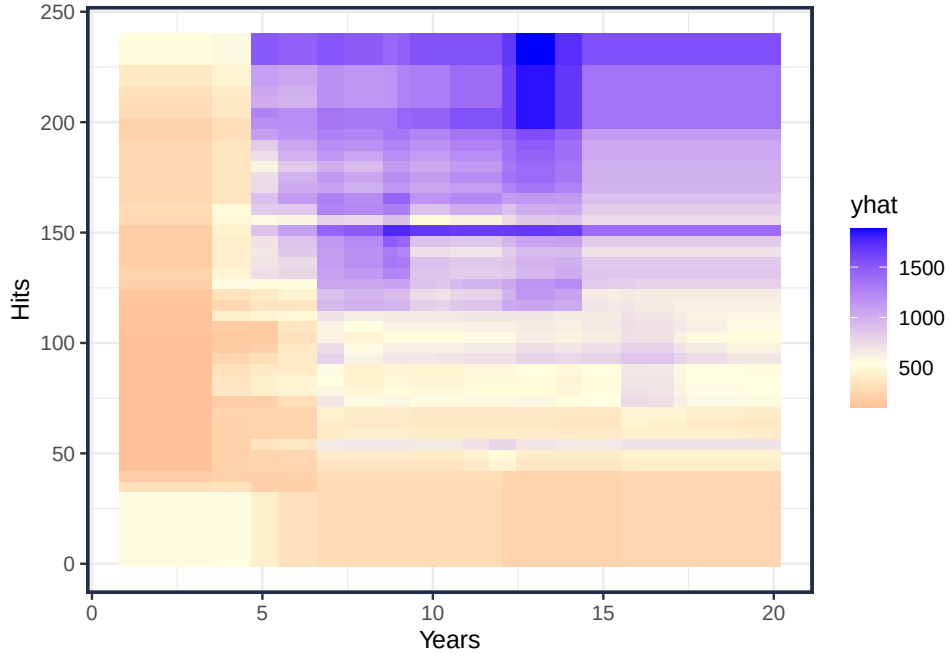


Figure 4: Average of bootstrap predictions

4 Random Forest

4.1 Random Forest

Random Forest is a modification of bagging that attempts to build *de-correlated trees* by considering a restricted set of features for splitting.

Algorithm 15.1 *Random Forest for Regression or Classification.*

1. For $b = 1$ to B:
   (a) Draw a bootstrap sample $\mathbf{Z}^*$ of size N from the training data.
   (b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.
      i. Select m variables at random from the p variables.
      ii. Pick the best variable/split-point among the m.
      iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.

To make a prediction at a new point x:

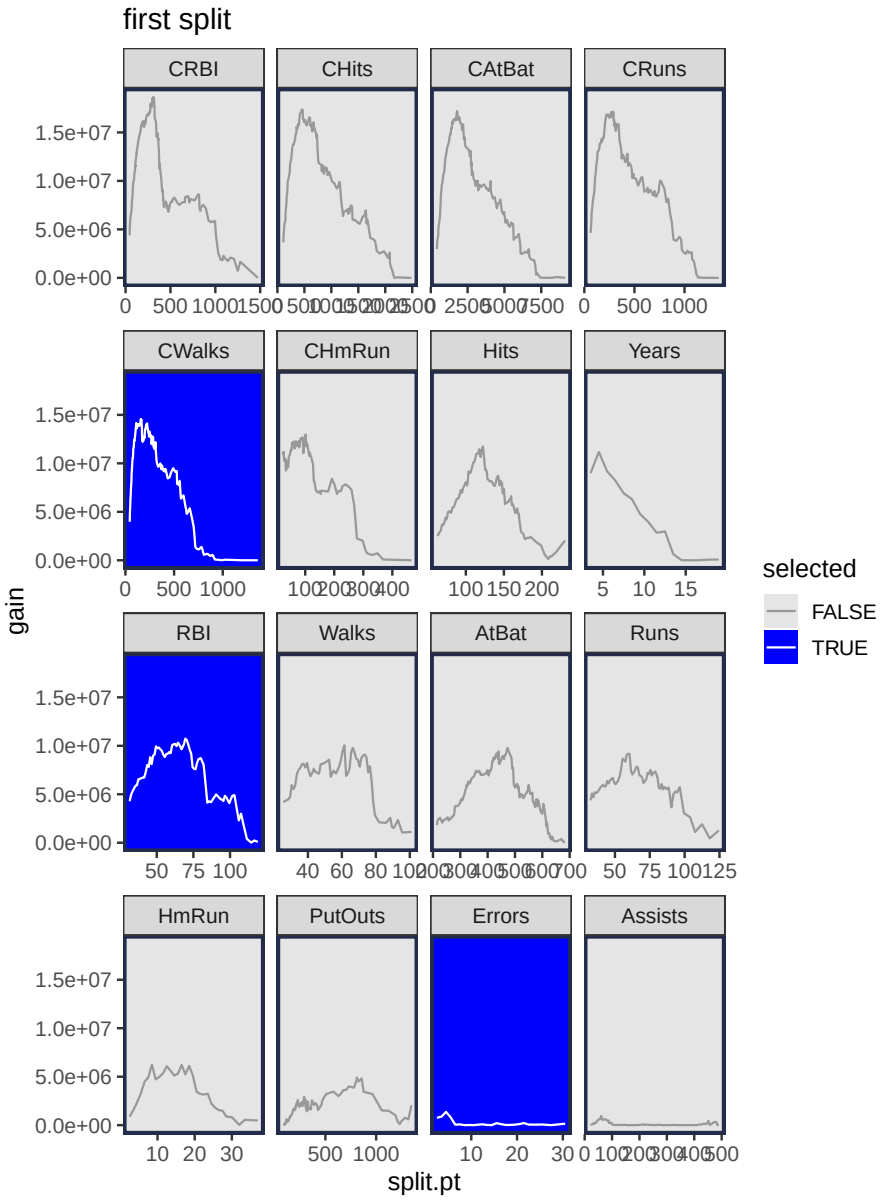
Regression: $\hat{f}_{\text{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^B T_b(x)$.

Classification: Let $\hat{C}_b(x)$ be the class prediction of the bth random-forest tree. Then $\hat{C}_{\text{rf}}^B(x) = \textit{majority vote}\ \{\hat{C}_b(x)\}_1^B$.

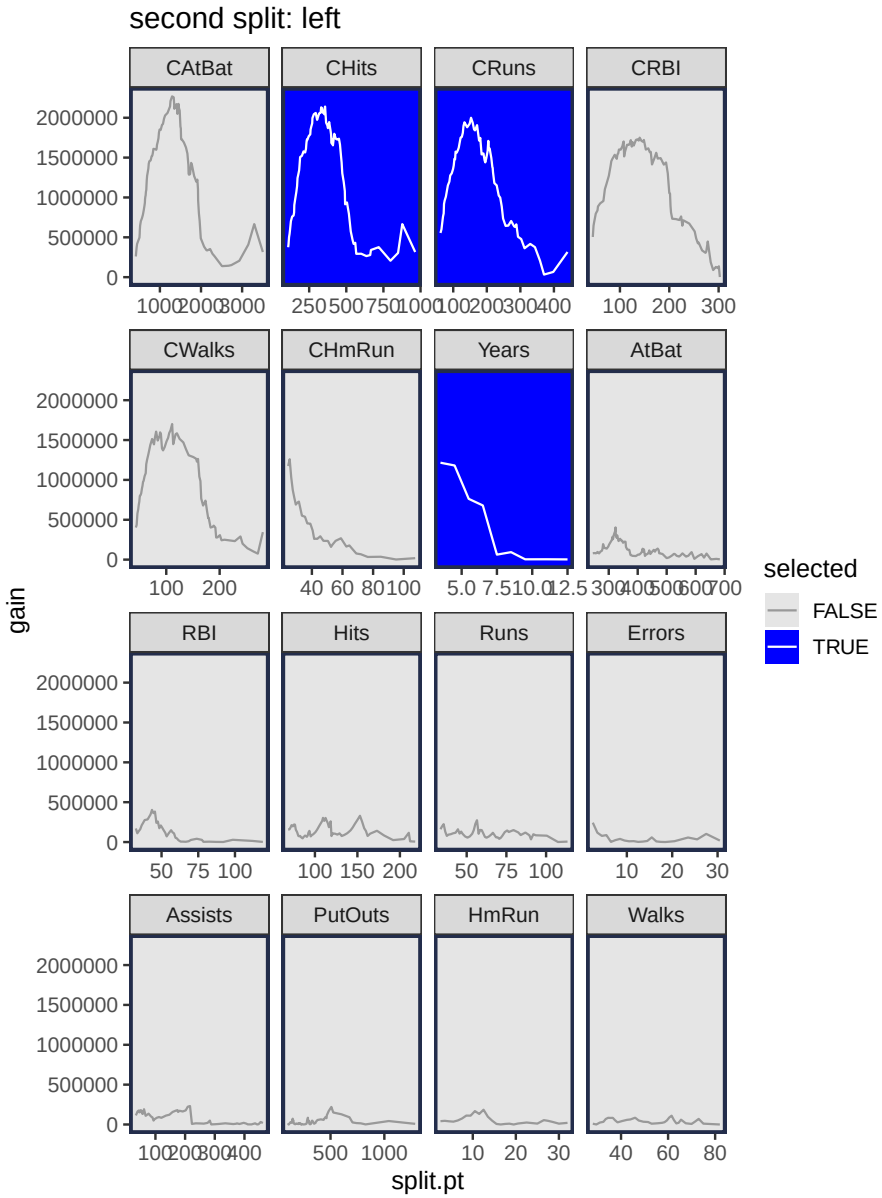
- Note: I recommend aggregating the probabilities for classification trees instead of majority vote.

4.1.1 Illustration of Restricted Set of Features for Splitting

var	split.pt	n.L	n.R	est.L	est.R	SSE.L	SSE.R	SSE	gain
CRBI	307.5	118	82	290.5	911.5	8010622	20282448	28293070	18653665
CHits	457.5	93	107	229.0	819.9	5299313	24273274	29572587	17374148
CAtBat	1779.5	95	105	236.8	824.0	5574003	24175441	29749444	17197291
CRuns	288.0	109	91	277.7	865.4	7452616	22368698	29821314	17125421
CWalks	**157.5**	**90**	**110**	**246.6**	**789.3**	**6768757**	**25600645**	**32369403**	**14577333**
CHmRun	101.5	156	44	409.9	1024.6	21520628	12459474	33980101	12966634
Hits	122.5	123	77	353.4	851.4	11979104	23225406	35204510	11742225
Years	4.5	69	131	219.5	716.6	5807172	29967467	35774639	11172096
RBI	**69.5**	**147**	**53**	**406.0**	**931.1**	**18079731**	**18126052**	**36205783**	**10740952**
Walks	61.5	155	45	424.4	961.0	19360871	17541643	36902514	10044222
AtBat	473.5	124	76	372.1	827.4	13829528	23351912	37181440	9765295
Runs	59.5	115	85	361.0	794.2	11945002	25829983	37774985	9171750
HmRun	8.5	94	106	357.6	711.4	12526389	28182028	40708417	6238318
PutOuts	809.0	186	14	502.1	1116.7	35199872	6828501	42028373	4918362
Errors	**4.5**	**71**	**129**	**433.5**	**606.5**	**7733767**	**37842076**	**45575843**	**1370892**
Assists	60.5	111	89	483.4	622.1	20984841	25011176	45996017	950719


first split
CRBI
CHits
CAtBat
CRuns
CWalks
CHmRun
Hits
Years
RBI
Walks
AtBat
Runs
HmRun
PutOuts
Errors
Assists
gain
split.pt
selected
FALSE
TRUE

var	split.pt	n.L	n.R	est.L	est.R	SSE.L	SSE.R	SSE	gain
CAtBat	1296.5	70	48	175.7	458.0	4142792	1599385	5742177	2268445
CHits	**357.0**	**77**	**41**	**192.2**	**475.1**	**4385523**	**1484232**	**5869754**	**2140868**
CRuns	**153.0**	**68**	**50**	**178.9**	**442.3**	**4202620**	**1808176**	**6010797**	**1999825**
CRBI	140.0	73	45	194.9	445.6	4612088	1648340	6260429	1750193
CWalks	111.0	74	44	197.9	446.2	4682658	1626914	6309572	1701050
CHmRun	25.5	73	45	209.4	422.1	4950482	1800792	6751274	1259349
Years	**3.5**	**48**	**70**	**168.0**	**374.6**	**4065840**	**2729314**	**6795153**	**1215469**
AtBat	323.5	50	68	222.3	340.7	4391102	3215583	7606685	403937
RBI	43.5	64	54	236.8	354.2	5514802	2092255	7607057	403565
Hits	153.0	105	13	271.9	440.8	7158659	522192	7680851	329771
Runs	56.5	77	41	255.4	356.5	5887826	1849136	7736962	273661
Errors	2.5	17	101	179.9	309.2	172699	7594684	7767383	243239
Assists	216.5	90	28	265.8	369.9	3189715	4589431	7779145	231477
PutOuts	508.0	107	11	276.7	425.3	7138760	651614	7790374	220248
HmRun	12.5	84	34	265.3	352.8	6395242	1429943	7825185	185437
Walks	61.5	103	15	278.9	370.7	7018267	881993	7900261	110361



4.2 Random Forest Tuning

There are two primary tuning parameters for Random Forest:

1. Variety: *mtry* controls the number of predictors that are evaluated for each split (this is named `max_fatures` scikit-learn)
2. Complexity: The depth/size of the trees are controlled by setting the *minimum number of observations in the leaf nodes* ($min.obs$) or the *depth* of the tree or the number of leaf nodes

Your Turn #2

How do these tuning parameters relate to the bias/variance trade-off?

- The tuning parameters can be determined from cross-validation or OOB error
- In `randomForest` and `ranger` packages:
  - For classification, the default value is $mtry = \lfloor \sqrt{p} \rfloor$ and $min.obs = 1$.
  - For regression, the default value is $mtry = \lfloor p/3 \rfloor$ and $min.obs = 5$.
- The *number of trees* is another tuning parameter, but want this to be as large as possible (subject to computational and memory constraints)
  - See This stats.stackexchange answer for further explanation.

4.3 OOB error

For each observation (x_i, y_i), construct its OOB prediction by averaging only those trees corresponding to bootstrap samples in which observation i *did not appear*.

$$\hat{f}(x_i) = \frac{1}{N_B(i)} \sum_{b=1}^{B} \mathbb{1}(x_i \in \mathrm{OOB(b)}) \cdot \mathrm{T}(\mathrm{x_i}; \hat{\theta}_\mathrm{b})$$

where $N_B(i)$ is the number of trees with observation i out-of-bag.

- Recall that there is a 37% chance that any observation is out-of-bag in any bootstrap sample.
- Thus, $N_B(i) \approx 0.37B$ (the number of trees used to estimate the OOB error is about 37% of the total number of trees in the forest).
  - More encouragement to use *many* trees in the forest

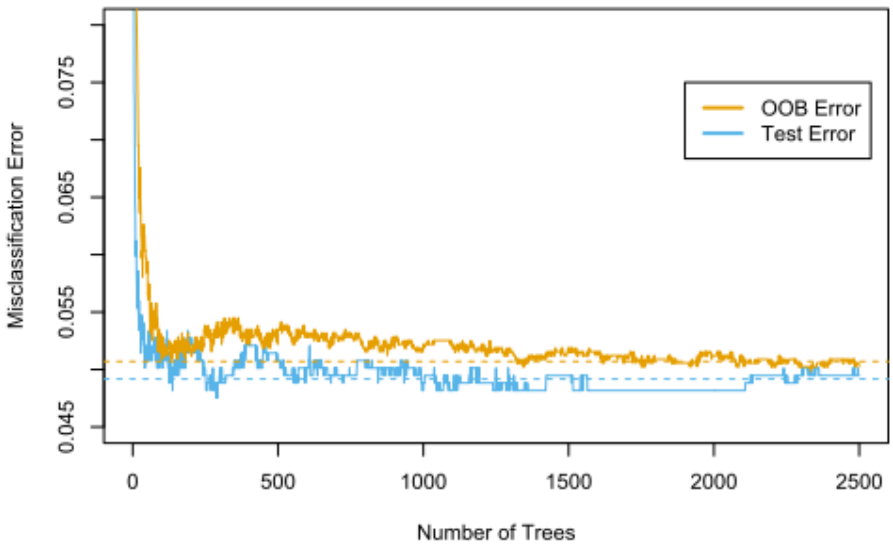


Figure 15.4 in ESL

4.4 Variable Importance

At each split in each tree, the improvement in the split-criterion is the importance measure attributed to the splitting variable, and is accumulated over all the trees in the forest separately for each variable.

The importance of predictor j in a single tree T:

$$\mathcal{I}_j(T) = \sum_t \text{gain}(t) \cdot \mathbb{1}(\text{split } t \text{ uses feature} j)$$

That is, the importance of feature j in tree T is the total *gain* from all splits involving feature j. In the equation, the sum is over all splits t in tree T.

The importance of predictor j in a forest is the average importance from all trees in the forest:

$$\mathcal{I}_j = \frac{1}{B} \sum_{b=1}^{B} \mathcal{I}_j(T_b)$$

- Note: a final normalizing step may transform importance scores to sum to 1
- There are other ways to measure feature importance, like permutation.

4.5 Random Forest and k-NN

Random Forests (especially with almost fully grown trees) are similar to k-NN methods, but they adaptively determines the neighbors instead of needing to pre-specify a distance metric.

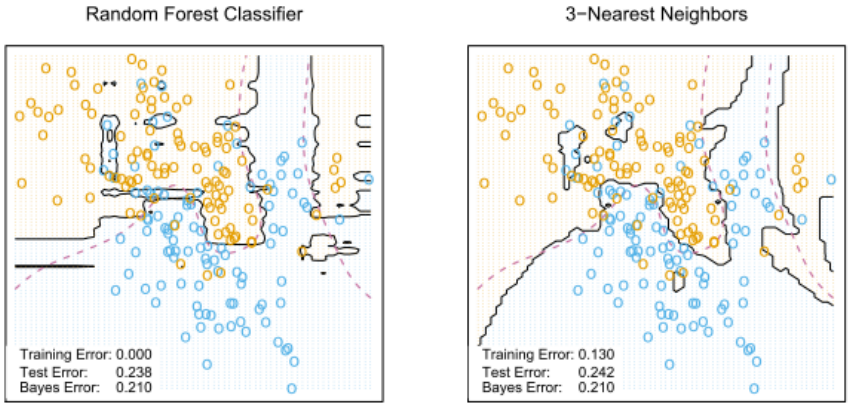


FIGURE 15.11. *Random forests versus 3-NN on the mixture data. The axis-oriented nature of the individual trees in a random forest lead to decision regions with an axis-oriented flavor.*

4.6 Random Forests in R

- `randomForest`
- `ranger`
- `randomForestSRC`
- `Rborist`
- `party`
- `aorsf` Oblique Random Forests
- `tidymodels` https://parsnip.tidymodels.org/reference/rand_forest.html